Let's assume around 10% of the users reach the next step and 90% of users drop off the flow before reaching the final step. We can also assume that no prefetching feature (prefetching the content before the user reaches the next step) is implemented. Figure 7.1 shows a rough estimation of what the QPS looks like for different steps. We know the final reservation TPS is 3 so we can work backward along the funnel. The QPS of the order confirmation page is 30 and the QPS for the detail page is 300.

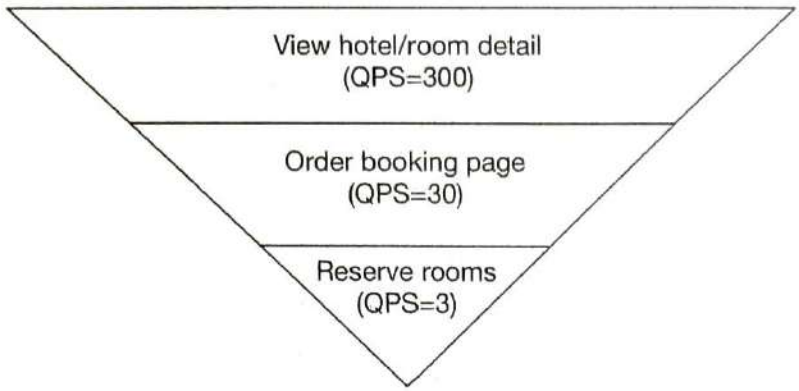


Figure 7.1: QPS distribution

Step 2 - Propose High-level Design and Get Buy-in

In this section, we'll discuss:

- API design
- Data models
- High-level design

API design

We explore the API design for the hotel reservation system. The most important APIs are listed below using the RESTful conventions.

Note that this chapter focuses on the design of a hotel reservation system. For a complete hotel website, the design needs to provide intuitive features for customers to search for rooms based on a large array of criteria. The APIs for these search features, while important, are not technically challenging. They are out of scope for this chapter.

Hotel-related APIs

API	Detail
GET /v1/hotels/ID	Get detailed information about a hotel.
POST /v1/hotels	Add a new hotel. This API is only available to hotel staff.
PUT /v1/hotels/ID	Update hotel information. This API is only available to hotel staff.
DELETE /v1/hotels/ID	Delete a hotel. This API is only available to hotel staff.

Table 7.1: Hotel-related APIs

Room-related APIs

API	Detail
GET /v1/hotels/ID/rooms/ID	Get detailed information about a room.
POST /v1/hotels/ID/rooms	Add a room. This API is only available to hotel staff.
PUT /v1/hotels/ID/rooms/ID	Update room information. This API is only available to hotel staff.
DELETE /v1/hotels/ID/rooms/ID	Delete a room. This API is only available to hotel staff.

Table 7.2: Hotel-related APIs

Reservation related APIs

API	Detail
GET /v1/reservations	Get the reservation history of the logged-in user.
GET /v1/reservations/ID	Get detailed information about a reservation.
POST /v1/reservations	Make a new reservation.
DELETE /v1/reservations/ID	Cancel a reservation.

Table 7.3: Reservation-related APIs

Making a new reservation is a very important feature. The request parameters of making a new reservation (`POST /v1/reservations`) could look like this.

```
{
  "startDate": "2021-04-28",
  "endDate":"2021-04-30",
  "hotelID":"245",
  "roomID":"U12354673389",
  "reservationID":"13422445"
}
```

Please note `reservationID` is used as the idempotency key to prevent double booking. Double booking means multiple reservations are made for the same room on the same day. The details are explained in the "Concurrency issue" section on page 206.

Data model

Before we decide which database to use, let's take a close look at the data access patterns. For the hotel reservation system, we need to support the following queries:

Query 1: View detailed information about a hotel.

Query 2: Find available types of rooms given a date range.

Query 3: Record a reservation.

Query 4: Look up a reservation or past history of reservations.

From the back-of-the-envelope estimation, we know the scale of the system is not large but we need to prepare for traffic surges during big events. With these requirements in mind, we choose a relational database because:

- A relational database works well with read-heavy and write less frequently workflow. This is because the number of users who visit the hotel website/apps is a few orders of magnitude higher than those who actually make reservations. NoSQL databases are generally optimized for writes and the relational database works well enough for read-heavy workflow.
- A relational database provides ACID (atomicity, consistency, isolation, durability) guarantees. ACID properties are important for a reservation system. Without those properties, it's not easy to prevent problems such as negative balance, double charge, double reservations, etc. ACID properties make application code a lot simpler and make the whole system easier to reason about. A relational database usually provides these guarantees.
- A relational database can easily model the data. The structure of the business data is very clear and the relationship between different entities (`hotel`, `room`, `room_type`, etc) is stable. This kind of data model is easily modeled by a relational database.

Now that we have chosen the relational database as our data store, let's explore the schema design. Figure 7.2 shows a straightforward schema design and it is the most natural way for many candidates to model the hotel reservation system.

Figure 7.2: Database schema

Most attributes are self-explanatory and we will only explain the `status` field in the reservation table. The `status` field can be in one of these states: `pending`, `paid`, `refunded`, `canceled`, `rejected`. The state machine is shown in Figure 7.3.

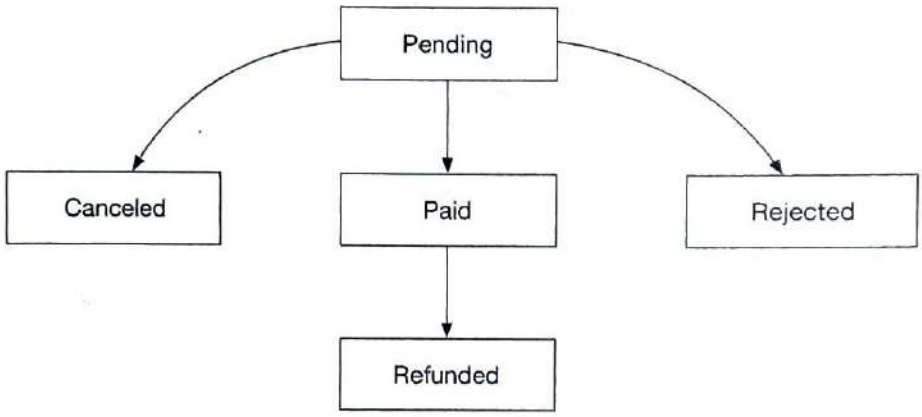


Figure 7.3: Reservation status

This schema design has a major issue. This data model works for companies like Airbnb as `room_id` (might be called `listing_id`) is given when users make reservations. However, this isn't the case for hotels. A user actually reserves **a type of room** in a given hotel instead of a specific room. For instance, a room type can be a standard room, king-size room, queen-size room with two queen beds, etc. Room numbers are given when the guest checks in and not at the time of the reservation. We need to update our data model to reflect this new requirement. See "Improved data model" in the deep dive section on

page 203 for more details.

High-level design

We use the microservice architecture for this hotel reservation system. Over the past few years, microservice architecture has gained great popularity. Companies that use microservice include Amazon, Netflix, Uber, Airbnb, Twitter, etc. If you want to learn more about the benefits of a microservice architecture, you can check out some good resources [1] [2] .

Our design is modeled with the microservice architecture and the high-level design diagram is shown in Figure 7.4.

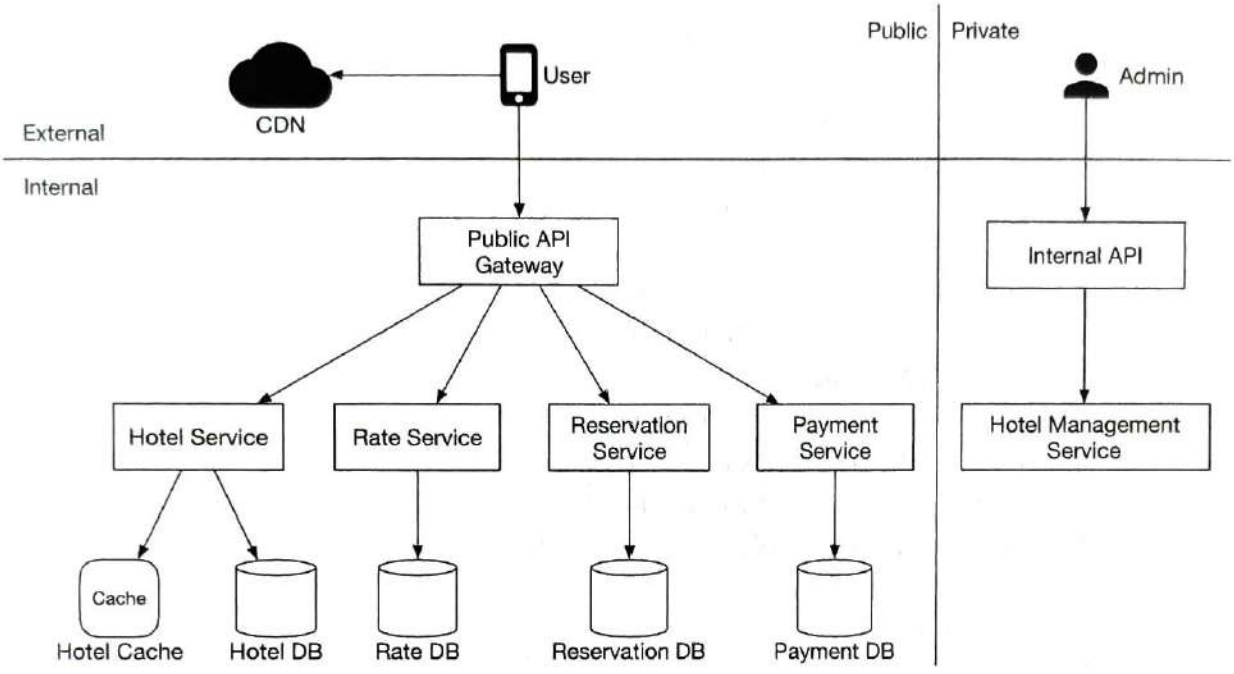


Figure 7.4: High-level design

We will briefly go over each component of the system from top to bottom.

- User: a user books a hotel room on their mobile phone or computer.
- Admin (hotel staff): authorized hotel staff perform administrative operations such as refunding a customer, canceling a reservation, updating room information, etc.
- CDN (content delivery network): for better load time, CDN is used to cache all static assets, including javascript bundles, images, videos, HTML, etc.
- Public API Gateway: this is a fully managed service that supports rate limiting, authentication, etc. The API gateway is configured to direct requests to specific services based on the endpoints. For example, requests to load the hotel homepage are directed to the hotel service and requests to book a hotel room are routed to the reservation service.
- Internal APIs: those APIs are only available for authorized hotel staff. They are accessible through internal software or websites. They are usually further protected

by a VPN (virtual private network).

- Hotel Service: this provides detailed information on hotels and rooms. Hotel and room data are generally static, so can be easily cached.
- Rate Service: this provides room rates for different future dates. An interesting fact about the hotel industry is that the price of a room depends on how full the hotel is expected to be for a given day.
- Reservation Service: receives reservation requests and reserves the hotel rooms. This service also tracks room inventory as rooms are reserved or reservations are canceled.
- Payment Service: executes payment from a customer and updates the reservation status to `paid` once a payment transaction succeeds, or `rejected` if the transaction fails.
- Hotel Management Service: only available to authorized hotel staff. Hotel staff are eligible to use the following features: view the record of an upcoming reservation, reserve a room for a customer, cancel a reservation, etc.

For clarity, Figure 7.4 omits many arrows of interactions between microservices. For example, as shown in Figure 7.5, there should be an arrow between Reservation service and Rate service. Reservation service queries Rate service for room rates. This is used to compute the total room charge for a reservation. Another example is that there should be many arrows connecting the Hotel Management Service with most of the other services. When an admin makes changes via Hotel Management Service, the requests are forwarded to the actual service owning the data, to handle the changes.

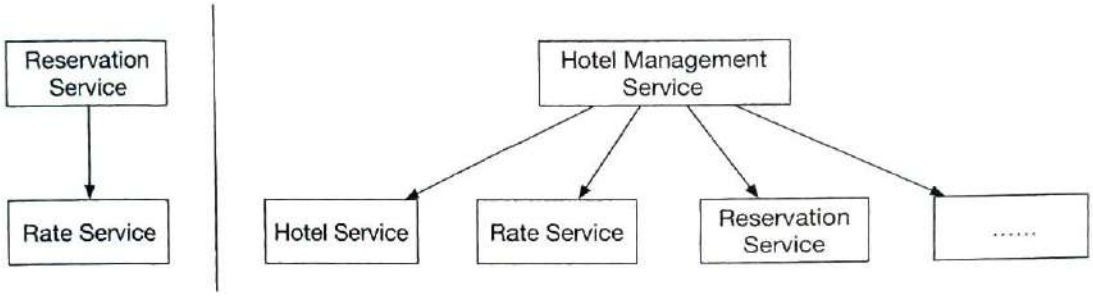


Figure 7.5: Connections between services

For production systems, inter-service communication often employs a modern and high-performance remote procedure call (RPC) framework like gPRC. There are many benefits to using such frameworks. To learn more about gPRC in particular, check out [3].

Step 3 - Design Deep Dive

Now we've talked about the high-level design, let's go deeper into the following.

- Improved data model
- Concurrency issues

- Scaling the system
- Resolving data inconsistency in the microservice architecture

Improved data model

As mentioned in the high-level design, when we reserve a hotel room, we actually reserve a type of room, as opposed to a specific room. What do we need to change about the API and schema to accommodate this?

For the reservation API, `roomID` is replaced by `roomTypeID` in the request parameter. The API to make a reservation looks like this:

`POST /v1/reservations`

Request parameters:

```
{
  "startDate": "2021-04-28",
  "endDate":"2021-04-30",
  "hotelID":"245",
  "roomTypeID":"12354673389",
  "reservationID":"13422445"
}
```

The updated schema is shown in Figure 7.6.

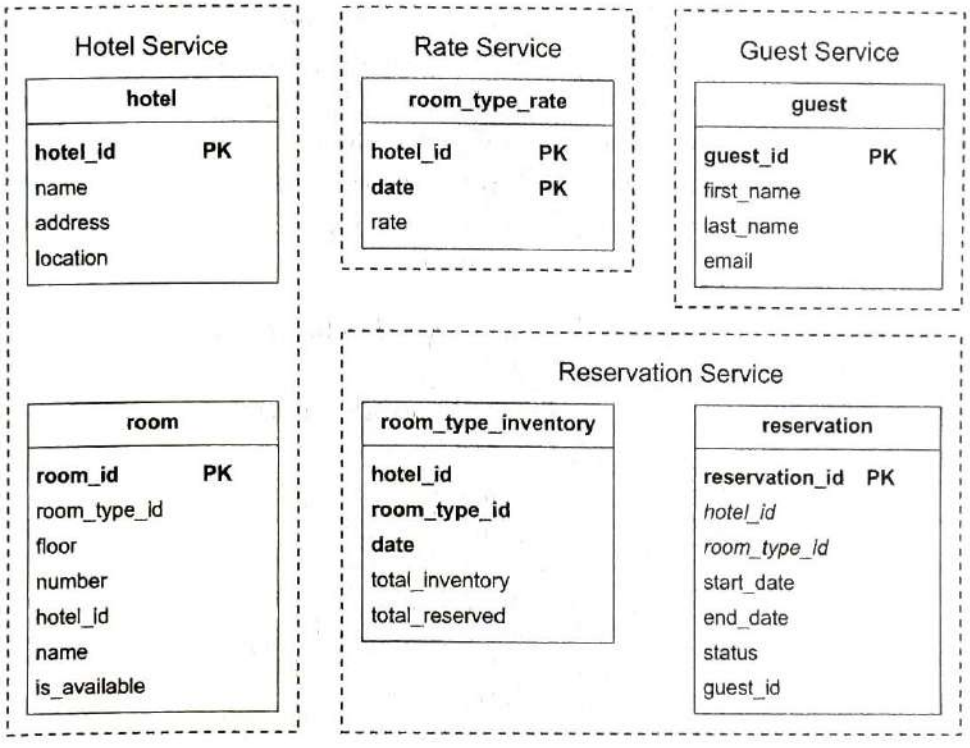


Figure 7.6: Updated schema

We'll briefly go over some of the most important tables.

room: contains information regarding a room.

room_type_rate: stores price data for a specific room type, for future dates.

reservation: records guest reservation data.

room_type_inventory: stores inventory data about hotel rooms. This table is very important for the reservation system, so let's take a close look at each column.

- hotel_id: ID of the hotel
- room_type_id: ID of a room type.
- date: a single date.
- total_inventory: the total number of rooms minus those that are temporarily taken off the inventory. Some rooms might be taken off from the market for maintenance.
- total_reserved: the total number of rooms booked for the specified hotel_id, room_type_id, and date.

There are other ways to design the room_type_inventory table, but having one row per date makes managing reservations within a date range and queries easy. As shown in Figure 7.6, (hotel_id, room_type_id, date) is the composite primary key. The rows of the table are pre-populated by querying the inventory data across all future dates within 2 years. We have a scheduled daily job that pre-populates inventory data when the dates advance further.

Now that we've finalized the schema design, let's do some estimation about the storage volume. As mentioned in the back-of-the-envelope estimation, we have 5,000 hotels. Assume each hotel has 20 types of rooms. That's (5,000 hotels $\times$20 types of rooms $\times$2 years $\times$365 days) = 73 million rows. 73 million is not a lot of data and a single database is enough to store the data. However, a single server means a single point of failure. To achieve high availability, we could set up database replications across multiple regions or availability zones.

Table 7.4 shows the sample data of the room_type_inventory table.

hotel_id	room_type_id	date	total_inventory	total_reserved
211	1001	2021-06-01	100	80
211	1001	2021-06-02	100	82
211	1001	2021-06-03	100	86
211	1001	...	...	
211	1001	2023-05-31	100	0
211	1002	2021-06-01	200	164
2210	101	2021-06-01	30	23
2210	101	2021-06-02	30	25

Table 7.4: Sample data of the room_type_inventory table

The room_type_inventory table is utilized to check if a customer can reserve a specific type of room or not. The input and output for a reservation might look like this:

- Input: `startDate` (2021-07-01), `endDate` (2021-07-03), `roomTypeId`, `hotelId`, `numberOfRoomsToReserve`
- Output: `True` if the specified type of room has inventory and users can book it. Otherwise, it returns `False`.

From the SQL perspective, it contains the following two steps:

1. Select rows within a date range

```
SELECT date, total_inventory, total_reserved
FROM room_type_inventory
WHERE room_type_id = ${roomTypeId} AND hotel_id = ${
hotelId}
AND date between ${startDate} and ${endDate}
```

   This query returns data like this:

date	total_inventory	total_reserved
2021-07-01	100	97
2021-07-02	100	96
2021-07-03	100	95

Table 7.5: Hotel inventory

2. For each entry, the application checks the condition below:

```
if ((total_reserved + ${numberOfRoomsToReserve}) <=
total_inventory)
```

If the condition returns `True` for all entries, it means there are enough rooms for each date within the date range.

One of the requirements is to support 10% overbooking. With the new schema, it is easy to implement:

```
if ((total_reserved + ${numberOfRoomsToReserve}) <= 110% *
total_inventory)
```

At this point, the interviewer might ask a follow-up question: "if the reservation data is too large for a single database, what would you do?" There are a few strategies:

- Store only current and future reservation data. Reservation history is not frequently accessed. So they can be archived and some can even be moved to cold storage.
- Database sharding. The most frequent queries include making a reservation or looking up a reservation by name. In both queries, we need to choose the hotel first, meaning `hotel_id` is a good sharding key. The data can be sharded by `hash(hotel_id) % number_of_servers`.

Concurrency issues

Another important problem to look at is double booking. We need to solve two problems:

1. The same user clicks on the book button multiple times.
2. Multiple users try to book the same room at the same time.

Let's take a look at the first scenario. As shown in Figure 7.7, two reservations are made.

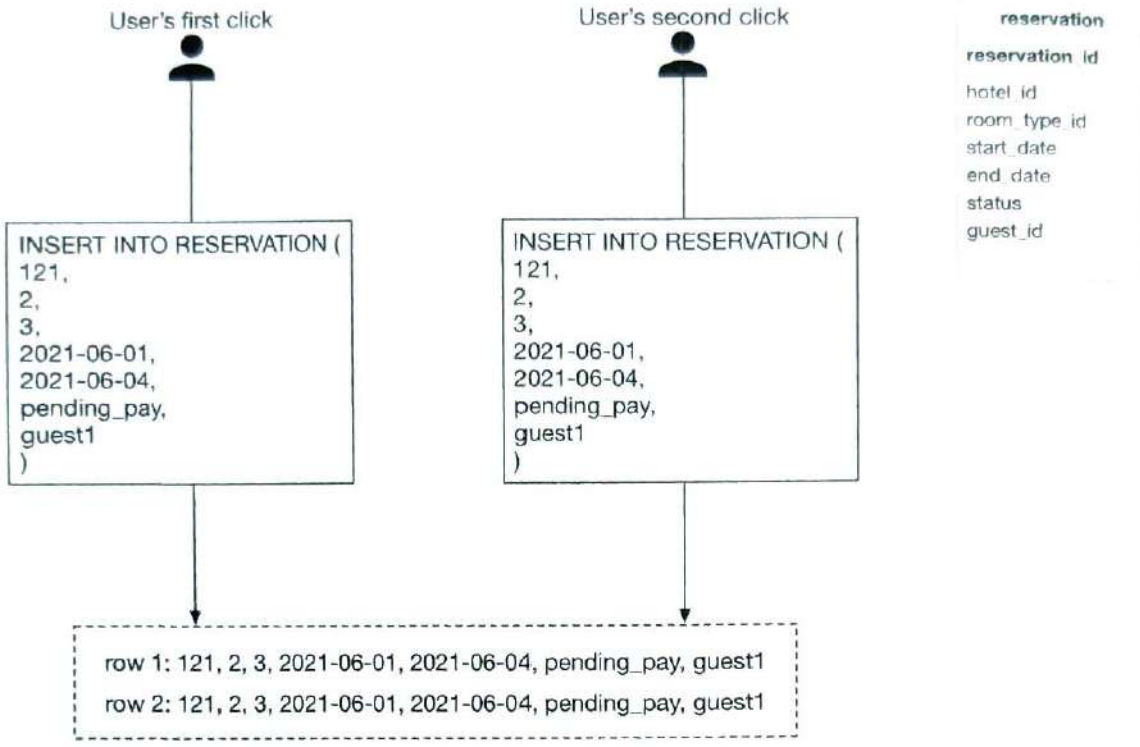


Figure 7.7: Two reservations are made

There are two common approaches to solve this problem:

- Client-side implementation. A client can gray out, hide or disable the "submit" button once a request is sent. This should prevent the double-clicking issue most of the time. However, this approach is not very reliable. For example, users can disable javascript, thereby bypassing the client check.
- Idempotent APIs. Add an idempotency key in the reservation API request. An API call is idempotent if it produces the same result no matter how many times it is called. Figure 7.8 shows how to use the idempotency key `reservation_id` to avoid the double-reservation issue. The detailed steps are explained below.

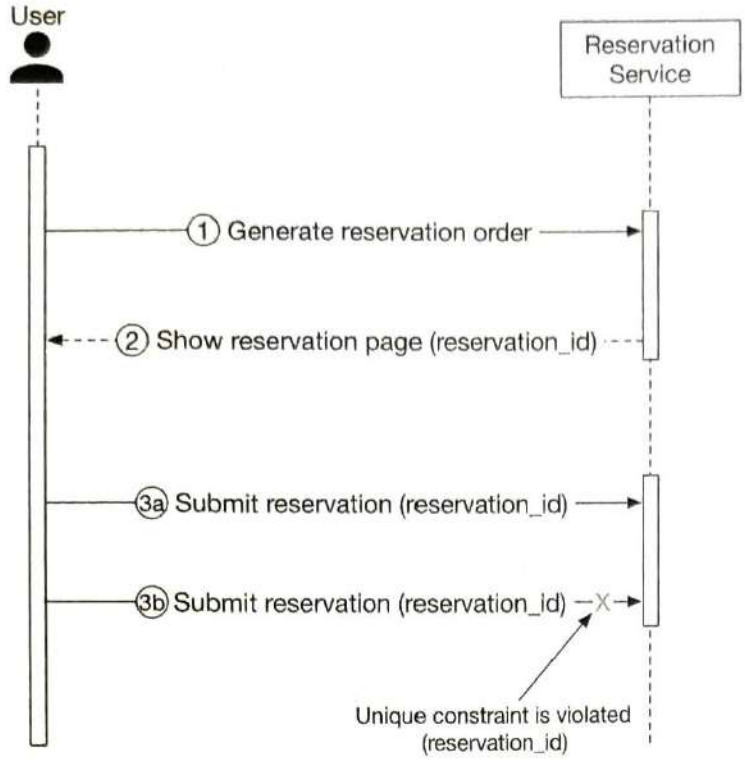


Figure 7.8: Unique constraint

1. Generate a reservation order. After a customer enters detailed information about the reservation (room type, check-in date, check-out date, etc) and clicks the "continue" button, a reservation order is generated by the reservation service.

2. The system generates a reservation order for the customer to review. The unique `reservation_id` is generated by a globally unique ID generator and returned as part of the API response. The UI of this step might look like this:

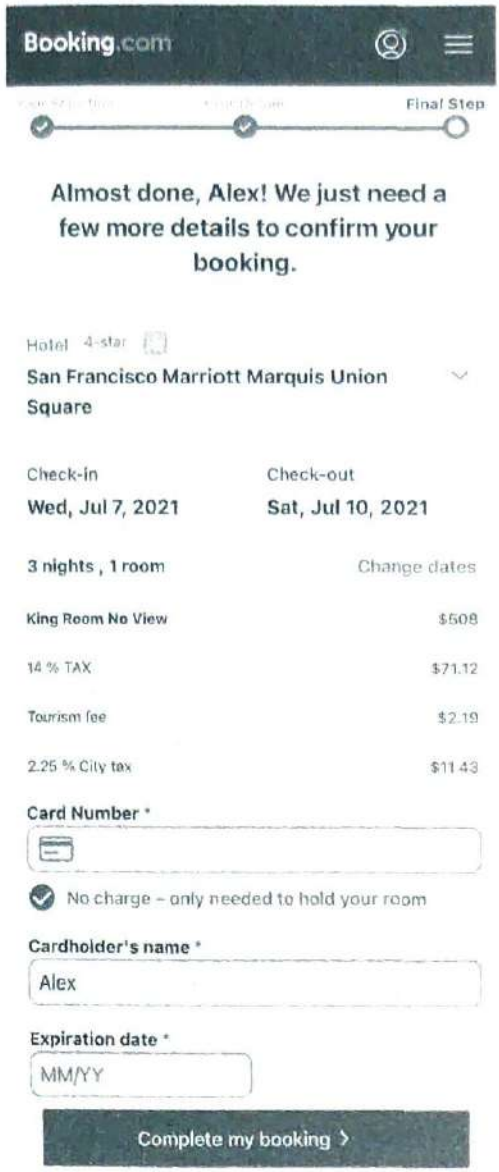


Figure 7.9: Confirmation page (Source: [4])

3a. Submit reservation 1. The `reservation_id` is included as part of the request. It is the primary key of the reservation table (Figure 7.6). Please note that the idempotency key doesn't have to be the `reservation_id`. We choose `reservation_id` because it already exists and works well for our design.

3b. If a user clicks the "Complete my booking" button a second time, reservation 2 is submitted. Because `reservation_id` is the primary key of the reservation table, we can rely on the unique constraint of the key to ensure no double reservation happens.

Figure 7.10 explains why double reservation can be avoided.

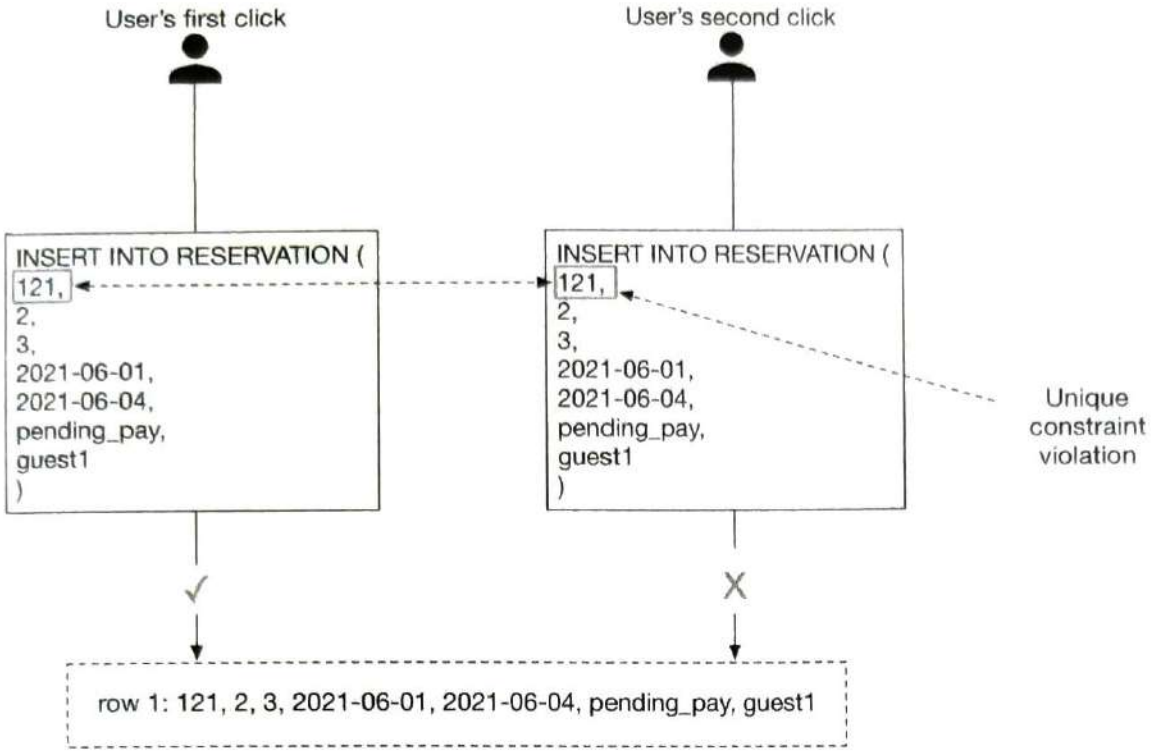


Figure 7.10: Unique constraint violation

Scenario 2: what happens if multiple users book the same type of room at the same time when there is only one room left? Let's consider the scenario as shown in Figure 7.11.

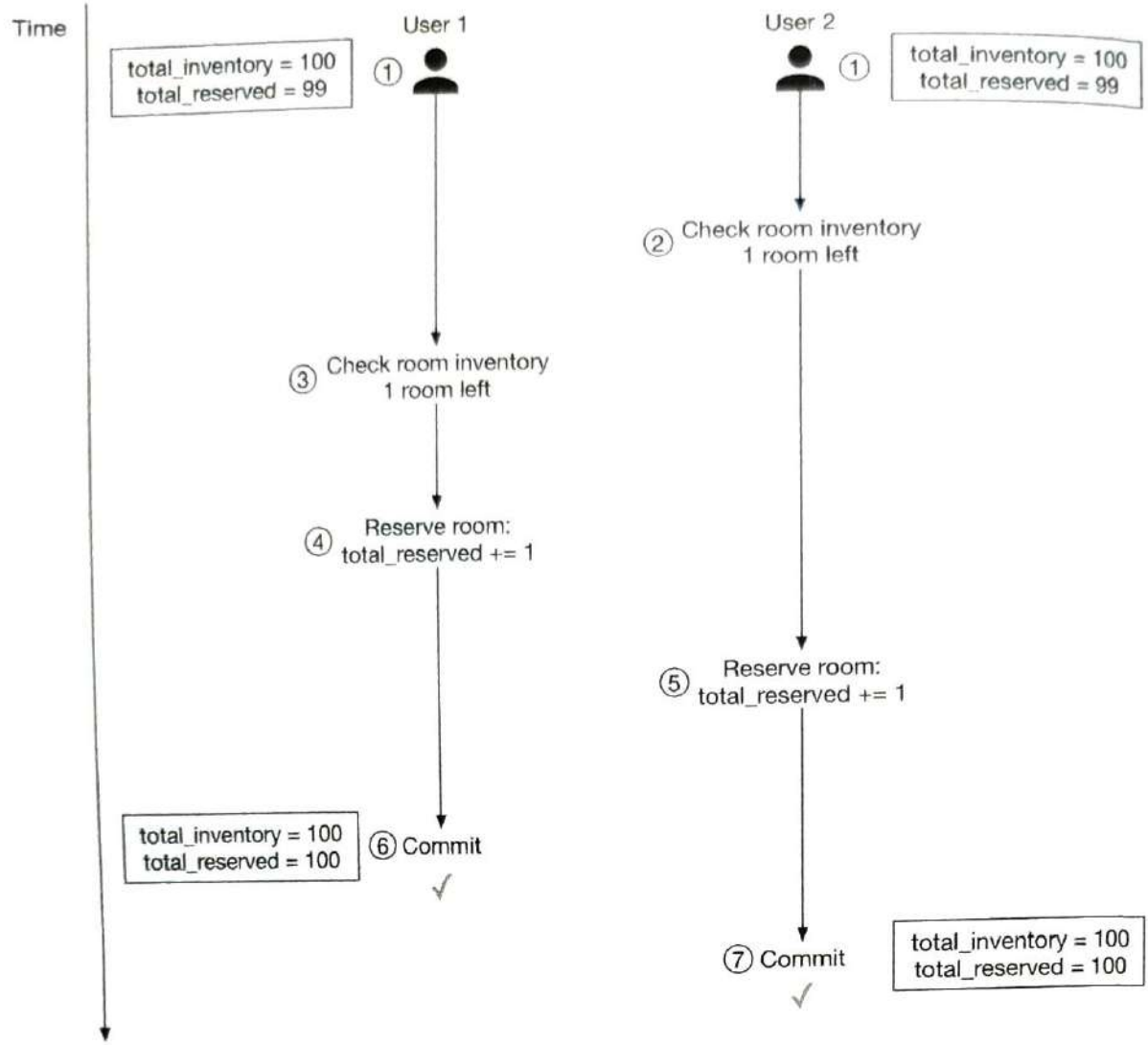


Figure 7.11: Race condition

1. Let's assume the database isolation level is not serializable [5]. User 1 and User 2 try to book the same type of room at the same time, but there is only 1 room left. Let's call User 1's execution transaction 1 and User 2's execution transaction 2. At this time, there are 100 rooms in the hotel and 99 of them are reserved.
2. Transaction 2 checks if there are enough rooms left by checking if (`total_reserved` + `rooms_to_book`) $\leq$ `total_inventory`. Since there is 1 more room left, it returns `True`.
3. Transaction 1 checks if there are enough rooms by checking if (`total_reserved` + `rooms_to_book`) $\leq$ `total_inventory`. Since there is 1 more room left, it returns `True` as well.
4. Transaction 1 reserves the room and updates the inventory: `reserved_room` becomes 100.
5. Then transaction 2 reserves the room. The **isolation** property in ACID means database transactions must complete their tasks independently from other transactions. So data changes made by transaction 1 are not visible to transaction 2 until transaction 1 is completed (committed). So transaction 2 still sees `total_reserved`

as 99 and reserves the room by updating the inventory: `reserved_room` becomes 100. This results in the system allowing both users to book a room, but there is only 1 room left.

6. Transaction 1 successfully commits the change.
7. Transaction 2 successfully commits the change.

The solution to this problem generally requires some form of locking mechanism. We explore the following techniques:

- Pessimistic locking
- Optimistic locking
- Database constraints

Before jumping into a fix, let's take a look at the SQL pseudo-code utilized to reserve a room. The SQL has two parts:

- Check room inventory
- Reserve a room

```
# step 1: check room inventory
SELECT date, total_inventory, total_reserved
FROM room_type_inventory
WHERE room_type_id = ${roomTypeId} AND hotel_id = ${hotelId}
AND date between ${startDate} and ${endDate}

# For every entry returned from step 1
if((total_reserved + ${numberOfRoomsToReserve}) > 110% *
  total_inventory) {
    Rollback
}

# step 2: reserve rooms
UPDATE room_type_inventory
SET total_reserved = total_reserved + ${numberOfRoomsToReserve}
WHERE room_type_id = ${roomTypeId}
AND date between ${startDate} and ${endDate}

Commit
```

Option 1: Pessimistic locking

The pessimistic locking [6], also called pessimistic concurrency control, prevents simultaneous updates by placing a lock on a record as soon as one user starts to update it. Other users who attempt to update the record have to wait until the first user has released the lock (committed the changes).

For MySQL, the "`SELECT ... FOR UPDATE`" statement works by locking the rows returned by a selection query. Let's assume a transaction is started by "transaction 1". Other

transactions have to wait for transaction 1 to finish before beginning another transaction. A detailed explanation is shown in Figure 7.12.

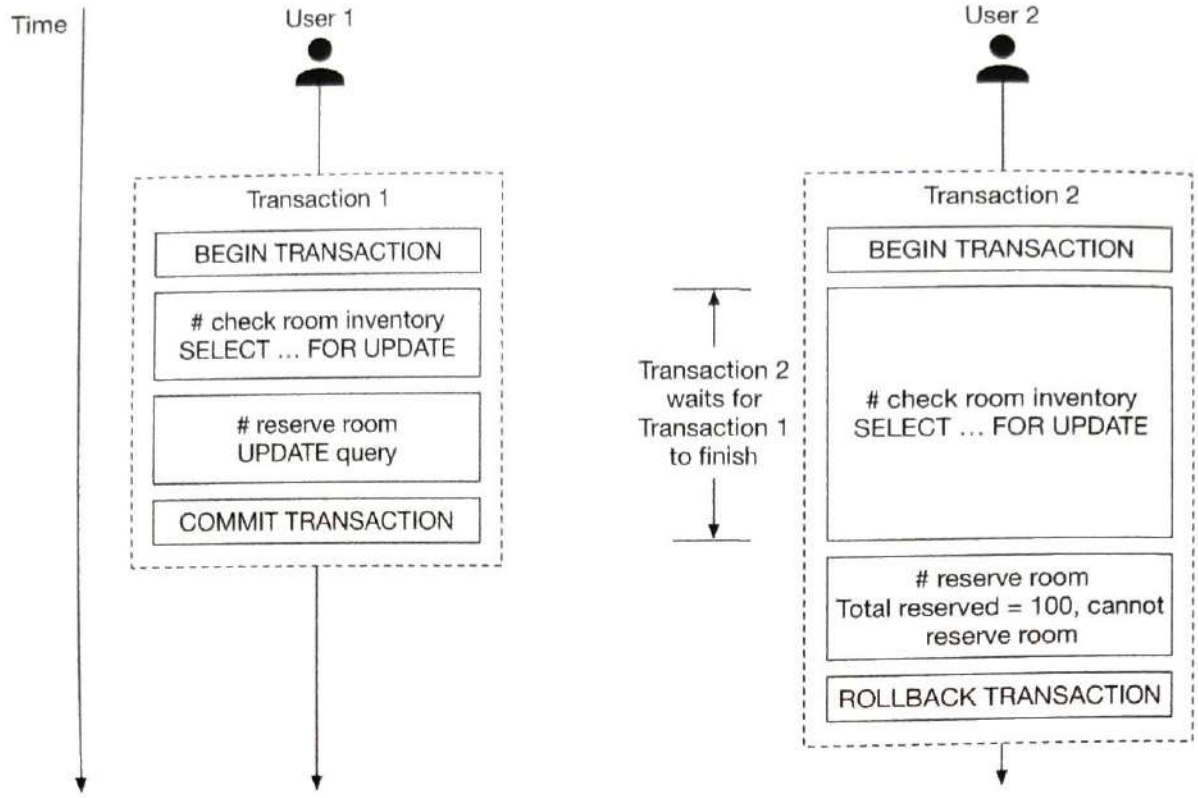


Figure 7.12: Pessimistic locking

In Figure 7.12, the "`SELECT ... FOR UPDATE`" statement of transaction 2 waits for transaction 1 to finish because transaction 1 locks the rows. After transaction 1 finishes, `total_reserved` becomes 100, which means there is no room for user 2 to book.

Pros:

- Prevents applications from updating data that is being or has been changed.
- It is easy to implement and it avoids conflict by serializing updates. Pessimistic locking is useful when data contention is heavy.

Cons:

- Deadlocks may occur when multiple resources are locked. Writing deadlock-free application code could be challenging.
- This approach is not scalable. If a transaction is locked for too long, other transactions cannot access the resource. This has a significant impact on database performance, especially when transactions are long-lived or involve a lot of entities.

Due to these limitations, we do not recommend pessimistic locking for the reservation system.

Option 2: Optimistic locking

Optimistic locking [7], also referred to as optimistic concurrency control, allows multiple concurrent users to attempt to update the same resource.

There are two common ways to implement optimistic locking: version number and timestamp. Version number is generally considered to be a better option because the server clock can be inaccurate over time. We explain how optimistic locking works with version number.

Figure 7.13 shows a successful case and a failure case.

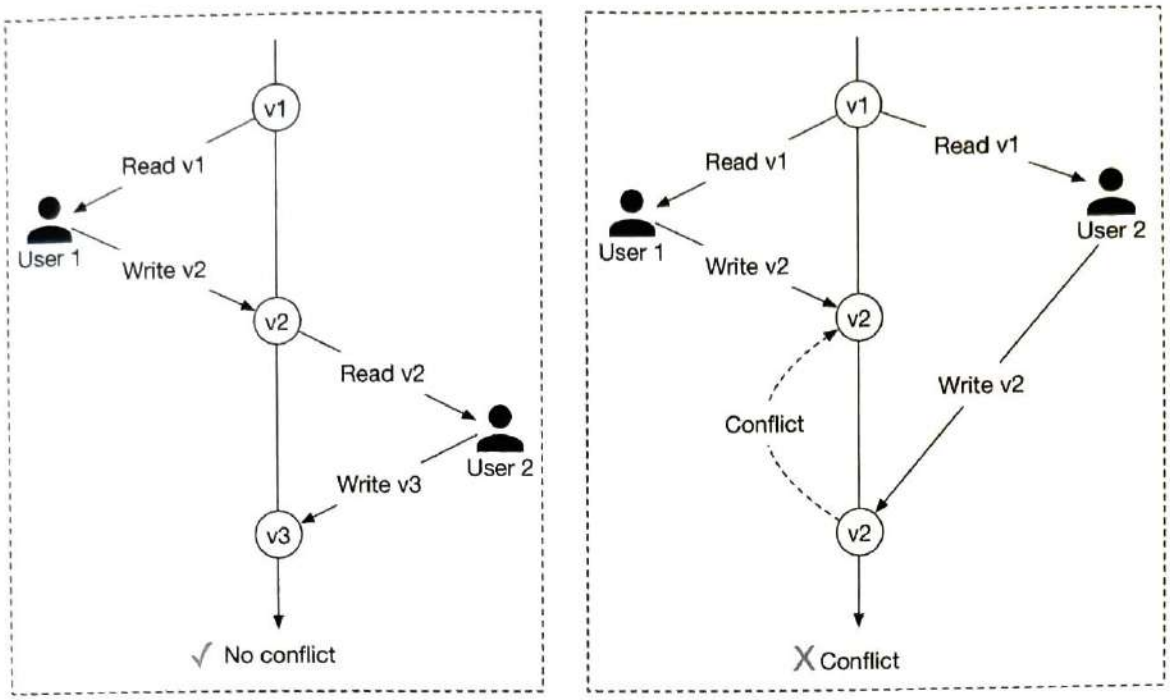


Figure 7.13: Optimistic locking

1. A new column called `version` is added to the database table.
2. Before a user modifies a database row, the application reads the version number of the row.
3. When the user updates the row, the application increases the version number by 1 and writes it back to the database.
4. A database validation check is put in place; the next version number should exceed the current version number by 1. The transaction aborts if the validation fails and the user tries again from step 2.

Optimistic locking is usually faster than pessimistic locking because we do not lock the database. However, the performance of optimistic locking drops dramatically when concurrency is high.

To understand why, consider the case when many clients try to reserve a hotel room at the same time. Because there is no limit on how many clients can read the available room

count, all of them read back the same available room count and the current version number. When different clients make reservations and write back the results to the database, only one of them will succeed, and the rest of the clients receive a version check failure message. These clients have to retry. In the subsequent round of retries, there is only one successful client again, and the rest have to retry. Although the end result is correct, repeated retries cause a very unpleasant user experience.

Pros:

- It prevents applications from editing stale data.
- We don't need to lock the database resource. There's actually no locking from the database point of view. It's entirely up to the application to handle the logic with the version number.
- Optimistic locking is generally used when the data contention is low. When conflicts are rare, transactions can complete without the expense of managing locks.

Cons:

- Performance is poor when data contention is heavy.

Optimistic locking is a good option for a hotel reservation system since the QPS for reservations is usually not high.

Option 3: Database constraints

This approach is very similar to optimistic locking. Let's explore how it works. In the `room_type_inventory` table, add the following constraint:

```
CONSTRAINT `check_room_count` CHECK((`total_inventory - total_reserved
  ` >= 0))
```

Using the same example as shown in Figure 7.14, when user 2 tries to reserve a room, `total_reserved` becomes 101, which violates the `total_inventory` (100)−`total_reserved` (101) ≥ 0 constraint. The transaction is then rolled back.

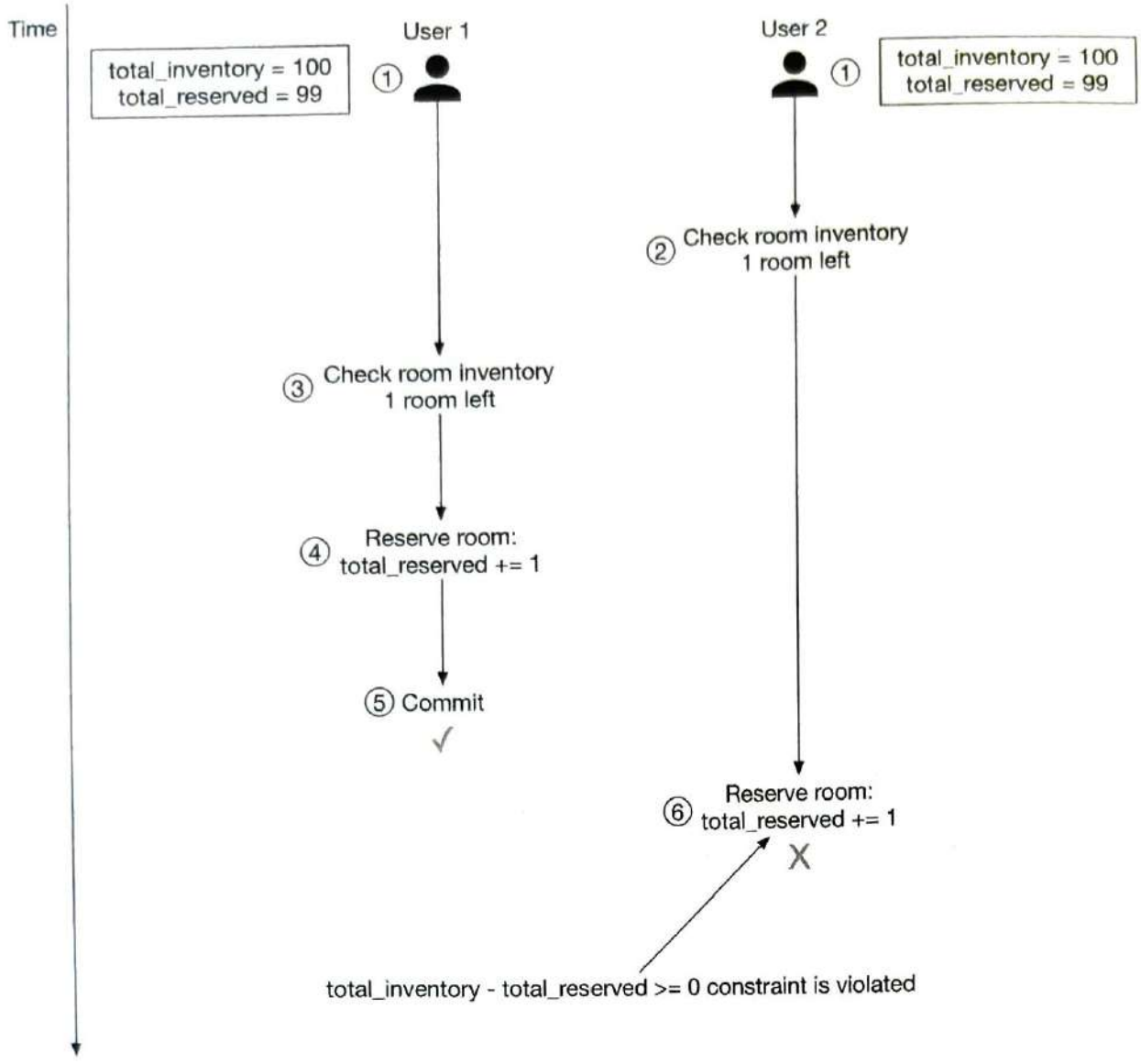


Figure 7.14: Database constraint

Pros

- Easy to implement.
- It works well when data contention is minimal.

Cons

- Similar to optimistic locking, when data contention is heavy, it can result in a high volume of failures. Users could see there are rooms available, but when they try to book one, they get the "no rooms available" response. The experience can be frustrating to users.
- The database constraints cannot be version-controlled easily like the application code.
- Not all databases support constraints. It might cause problems when we migrate from one database solution to another.

Since this approach is easy to implement and the data contention for a hotel reservation is usually not high (low QPS), it is another good option for the hotel reservation system.

Scalability

Usually, the load of the hotel reservation system is not high. However, the interviewer might have a follow-up question: "what if the hotel reservation system is used not just for a hotel chain but for a popular travel site such as booking.com or expedia.com?" In this case, the QPS could be 1,000 times higher.

When the system load is high, we need to understand what might become the bottleneck. All our services are stateless, so they can be easily expanded by adding more servers. The database, however, contains all the states and cannot be scaled up by simply adding more databases. Let's explore how to scale the database.

Database sharding

One way to scale the database is to apply database sharding. The idea is to split the data into multiple databases so that each of them only contains a portion of data.

When we shard a database, we need to consider how to distribute the data. As we can see from the data model section, most queries need to filter by `hotel_id`. So a natural conclusion is we shard data by `hotel_id`. In Figure 7.15, the load is spread among 16 shards. Assume the QPS is 30,000. After database sharding, each shard handles $\frac{30,000}{16} = 1,875$ QPS, which is within a single MySQL server's load capacity.

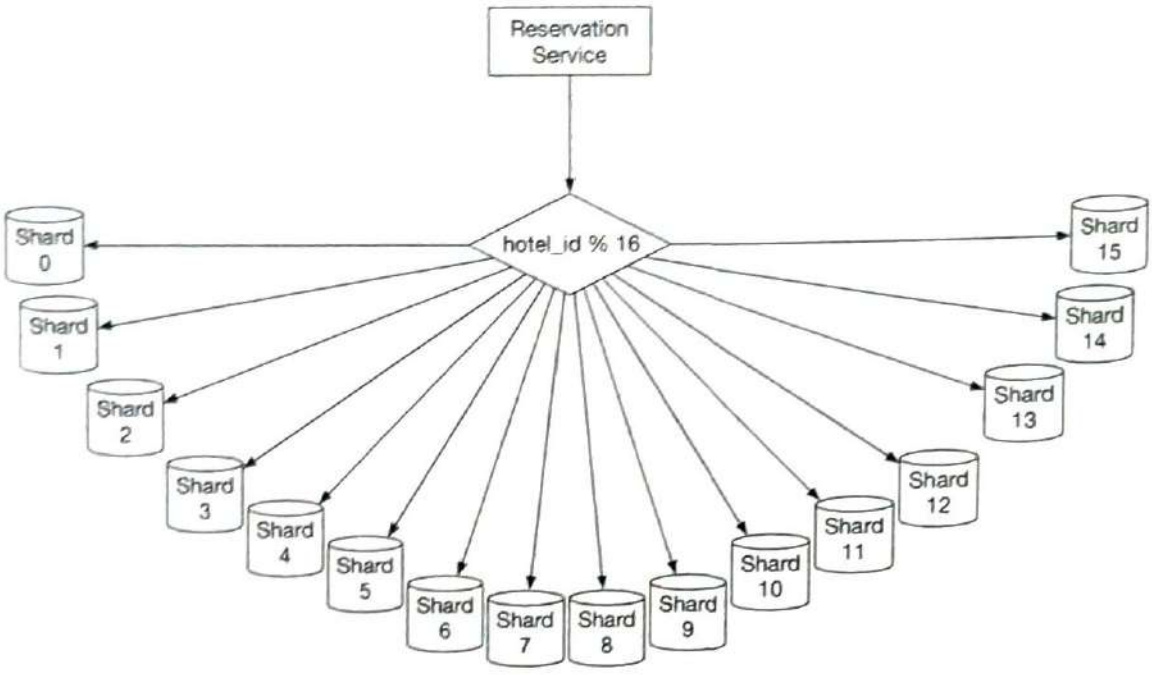


Figure 7.15: Database sharding

Caching

The hotel inventory data has an interesting characteristic; only current and future hotel inventory data are meaningful because customers can only book rooms in the near

future.

So for the storage choice, ideally we want to have a time-to-live (TTL) mechanism to expire old data automatically. Historical data can be queried on a different database. Redis is a good choice because TTL and Least Recently Used (LRU) cache eviction policy help us make optimal use of memory.

If the loading speed and database scalability become an issue (for instance, we are designing at booking.com or Expedia's scale), we can add a cache layer on top of the database and move the check room inventory and reserve room logic to the cache layer, as shown in Figure 7.16. In this design, only a small percentage of the requests hit the inventory database as most ineligible requests are blocked by the inventory cache. One thing worth mentioning is that even when there is enough inventory shown in Redis, we still need to recheck the inventory at the database side as a precaution. The database is the source of truth for the inventory data.

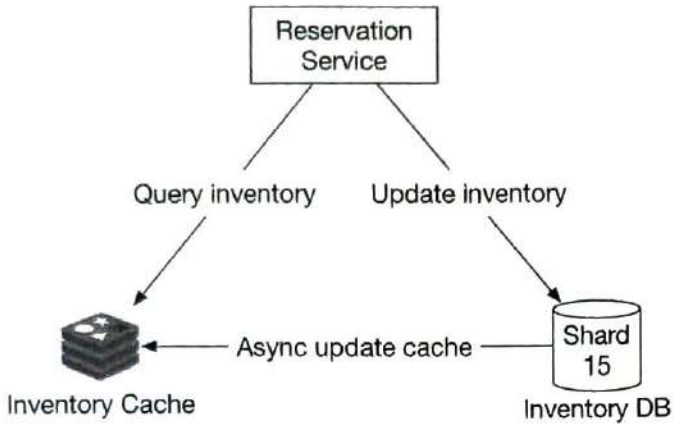


Figure 7.16: Caching

Let's first go over each component in this system.

Reservation service: supports the following inventory management APIs:

- Query the number of available rooms for a given hotel, room type, and date range.
- Reserve a room by executing `total_reserved` +1.
- Update inventory when a user cancels a reservation.

Inventory cache: all inventory management query operations are moved to the inventory cache (Redis) and we need to pre-populate inventory data to the cache. The cache is a key-value store with the following structure:

```
key: hotelID_roomTypeID_{date}
value: the number of available rooms for the given hotel ID,
  room type ID and date.
```

For a hotel reservation system, the volume of read operations (check room inventory) is an order of magnitude higher than write operations. Most of the read operations are

answered by the cache.

Inventory DB: stores inventory data as the source of truth.

New challenges posed by the cache

Adding a cache layer significantly increases the system scalability and throughput, but it also introduces a new challenge: how to maintain data consistency between the database and the cache.

When a user books a room, two operations are executed in the happy path:

1. Query room inventory to find out if there are enough rooms left. The query runs on the Inventory cache.
2. Update inventory data. The inventory DB is updated first. The change is then propagated to the cache asynchronously. This asynchronous cache update could be invoked by the application code, which updates the inventory cache after data is saved to the database. It could also be propagated using change data capture (CDC) [8]. CDC is a mechanism that reads data changes from the database and applies the changes to another data system. One common solution is Debezium [9]. It uses a source connector to read changes from a database and applies them to cache solutions such as Redis [10].

Because the inventory data is updated on the database first, there is a possibility that the cache does not reflect the latest inventory data. For example, the cache may report there is still an empty room when the database says there is no room left or vice versa.

If you think carefully, you find that the inconsistency between inventory cache and database actually does not matter, as long as the database does the final inventory validation check.

Let's take a look at an example. Let's say the cache says there is still an empty room, but the database says no. In this case, when the user queries the room inventory, they find there is still room available, so they try to reserve it. When the request reaches the inventory database, the database does the validation and finds that there is no room left. In this case, the client receives an error, indicating someone else just booked the last room before them. When a user refreshes the website, they probably see there is no room left because the database has synchronized inventory data to the cache, before they click the refresh button.

Pros

- Reduced database load. Since read queries are answered by the cache layer, database load is significantly reduced.
- High performance. Read queries are very fast because results are fetched from memory.

Cons

- Maintaining data consistency between the database and cache is hard. We need to think carefully about how this inconsistency affects user experience.

Data consistency among services

In a traditional monolithic architecture [11], a shared relational database is used to ensure data consistency. In our microservice design, we chose a hybrid approach by having Reservation Service handle both reservation and inventory APIs so that the inventory and reservation database tables are stored in the same relational database. As explained in the "Concurrency issues" section on page 206, this arrangement allows us to leverage the ACID properties of the relational database to elegantly handle many concurrency issues that arise during the reservation flow.

However, if your interviewer is a microservice purist, they might challenge this hybrid approach. In their mind, for a microservice architecture, each microservice has its own databases as shown on the right in Figure 7.17.

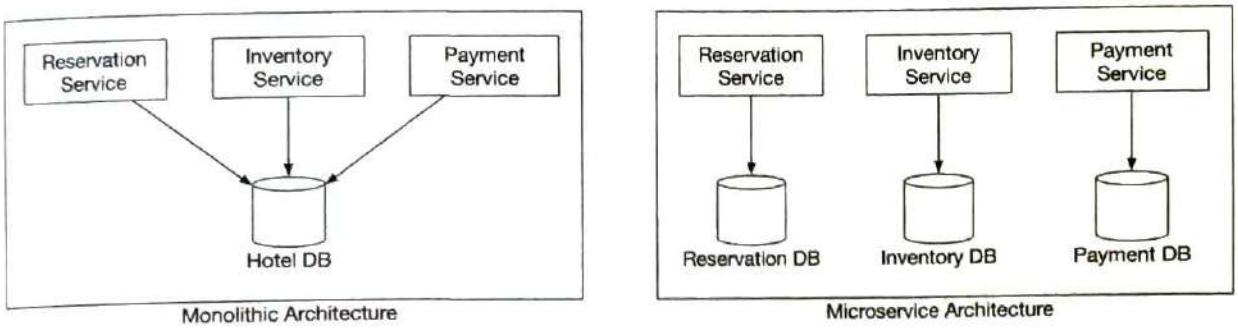


Figure 7.17: Monolithic vs microservice

This pure design introduces many data consistency issues. Since this is the first time we cover microservices, let's explain how and why it happens. To make it easier to understand, only two services are used in this discussion. In the real world, there could be hundreds of microservices within a company. In a monolithic architecture, as shown in Figure 7.18, different operations can be wrapped in a single transaction to ensure ACID properties.

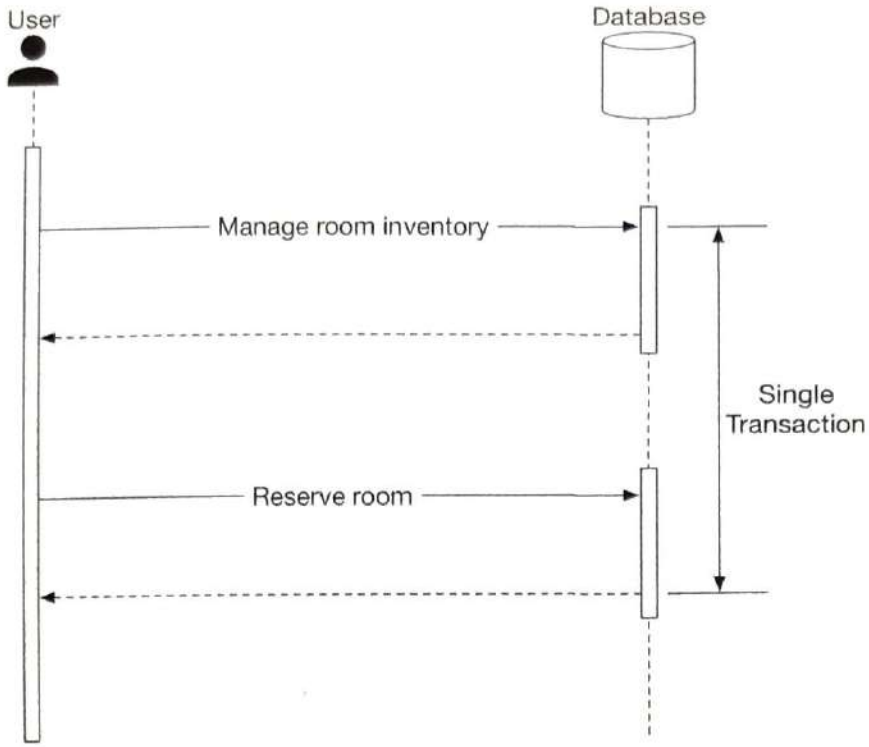


Figure 7.18: Monolithic architecture

However, in a microservice architecture, each service has its own database. One logically atomic operation can span multiple services. This means we cannot use a single transaction to ensure data consistency. As shown in Figure 7.19, if the update operation fails in the reservation database, we need to roll back the reserved room count in the inventory database. Generally, there is only one happy path, but many failure cases that could cause data inconsistency.

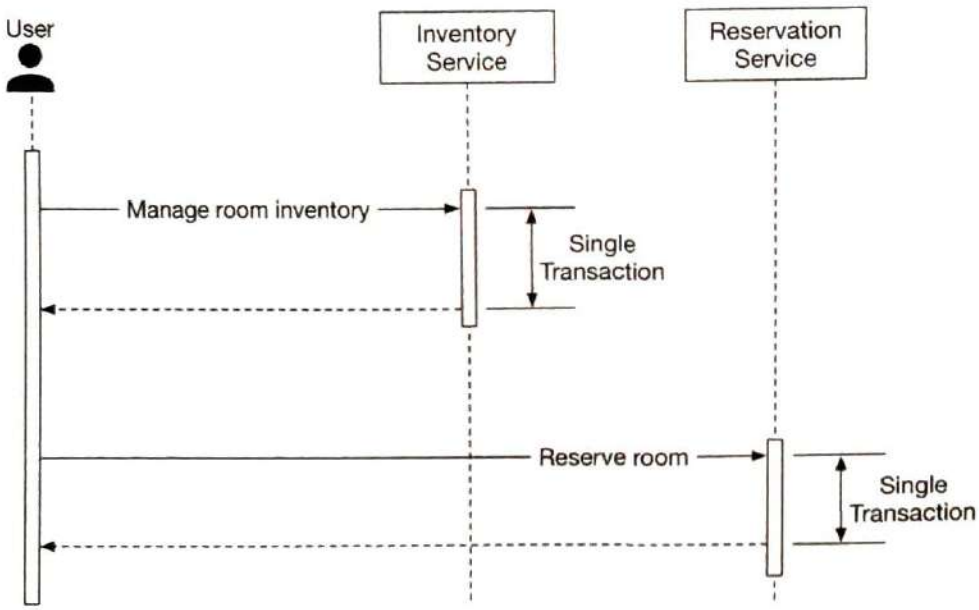


Figure 7.19: Microservice architecture

To address the data inconsistency, here is a high-level summary of industry-proven tech-

niques. If you want to read the details, please refer to the reference materials.

- Two-phase commit (2PC) [12]. 2PC is a database protocol used to guarantee atomic transaction commit across multiple nodes, i.e., either all nodes succeeded or all nodes failed. Because 2PC is a blocking protocol, a single node failure blocks the progress until the node has recovered. It's not performant.
- Saga. A Saga is a sequence of local transactions. Each transaction updates and publishes a message to trigger the next transaction step. If a step fails, the saga executes compensating transactions to undo the changes that were made by preceding transactions [13]. 2PC works as a single commit to perform ACID transactions while Saga consists of multiple steps and relies on eventual consistency.

It is worth noting that addressing data inconsistency between microservices requires some complicated mechanisms that greatly increase the complexity of the overall design. It is up to you as an architect to decide if the added complexity is worth it. For this problem, we decided that it was not worth it and so went with the more pragmatic approach of storing reservation and inventory data under the same relational database.

Step 4 - Wrap Up

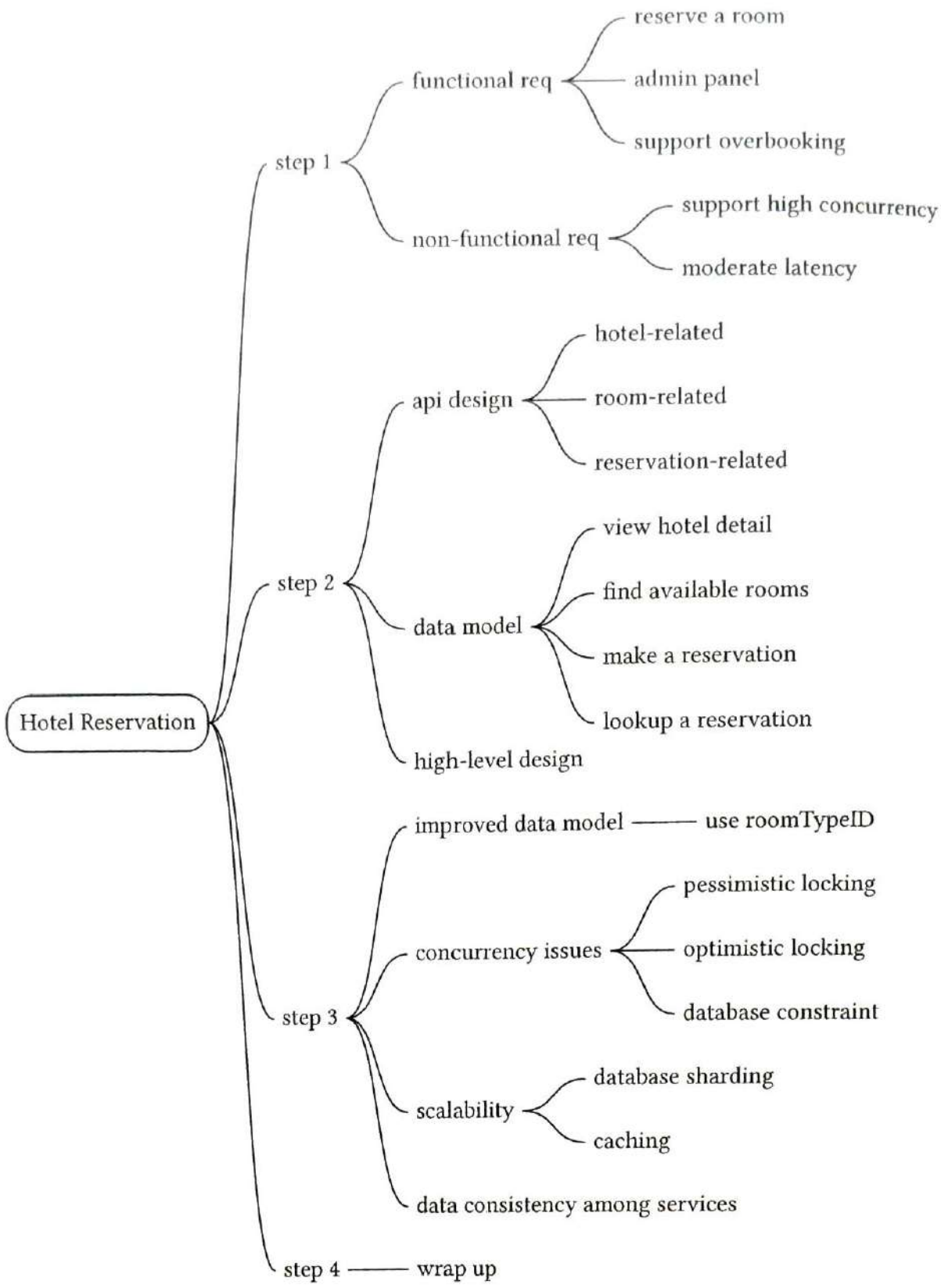
In this chapter, we presented a design for a hotel reservation system. We started by gathering requirements and calculating a back-of-the-envelope estimation to understand the scale. In the high-level design, we presented the API design, the first draft of the data model, and the system architecture diagram. In the deep dive, we explored alternative database schema design as we realized reservations should be made at the room type level, as opposed to specific rooms. We discussed race conditions in depth and proposed a few potential solutions:

- pessimistic locking
- optimistic locking
- database constraints

We then discussed different approaches to scale the system, including database sharding and using Redis cache. Lastly, we went through data consistency issues in microservice architecture and briefly went through a few solutions.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Chapter Summary



Reference Material

[1] What Are The Benefits of Microservices Architecture? https://www.appdynamics.com/topics/benefits-of-microservices.

[2] Microservices. https://en.wikipedia.org/wiki/Microservices.

[3] gRPC. https://www.grpc.io/docs/what-is-grpc/introduction/.

[4] Booking.com iOS app.

[5] Serializability. https://en.wikipedia.org/wiki/Serializability.

[6] Optimistic and pessimistic record locking. https://ibm.co/3Eb293O.

[7] Optimistic concurrency control. https://en.wikipedia.org/wiki/Optimistic_concurrency_control.

[8] Change data capture. https://docs.oracle.com/cd/B10500_01/server.920/a96520/cdc.htm.

[9] Debizium. https://debezium.io/.

[10] Redis sink. https://bit.ly/3r3AEUD.

[11] Monolithic Architecture. https://microservices.io/patterns/monolithic.html.

[12] Two-phase commit protocol. https://en.wikipedia.org/wiki/Two-phase_commit_protocol.

[13] Saga. https://microservices.io/patterns/data/saga.html.

8 Distributed Email Service

In this chapter, we design a large-scale email service, such as Gmail, Outlook, or Yahoo Mail. The growth of the internet has led to an explosion in the volume of emails. In 2020, Gmail had over 1.8 billion active users and Outlook had over 400 million users worldwide [1] [2].

Figure 8.1: Popular email providers

Step 1 - Understand the Problem and Establish Design Scope

Over the years, email services have changed significantly in complexity and scale. A modern email service is a complex system with many features. There is no way we can design a real-world system in 45 minutes. So before jumping into the design, we definitely want to ask clarifying questions to narrow down the scope.

Candidate: How many people use the product?
Interviewer: One billion users.

Candidate: I think the following features are important:

- Authentication.
- Send and receive emails.
- Fetch all emails.
- Filter emails by read and unread status.
- Search emails by subject, sender, and body.
- Anti-spam and anti-virus.

Interviewer: That's a good list. We don't need to worry about authentication. Let's focus on the other features you mentioned.

Candidate: How do users connect with mail servers?
Interviewer: Traditionally, users connect with mail servers through native clients that use SMTP, POP, IMAP, and vendor-specific protocols. Those protocols are legacy to some extent, yet still very popular. For this interview, let's assume HTTP is used for client and server communication.

Candidate: Can emails have attachments?
Interviewer: Yes.

Non-functional requirements

Next, let's go over the most important non-functional requirements.

Reliability. We should not lose email data.

Availability. Email and user data should be automatically replicated across multiple nodes to ensure availability. Besides, the system should continue to function despite partial system failures.

Scalability. As the number of users grows, the system should be able to handle the increasing number of users and emails. The performance of the system should not degrade with more users or emails.

Flexibility and extensibility. A flexible/extensible system allows us to add new features or improve performance easily by adding new components. Traditional email protocols such as POP and IMAP have very limited functionality (more on this in high-level design). Therefore, we may need custom protocols to satisfy the flexibility and extensibility requirements.

Back-of-the-envelope estimation

Let's do a back-of-the-envelope calculation to determine the scale and to discover some challenges our solution will need to address. By design, emails are storage heavy applications.

- 1 billion users.
- Assume the average number of emails a person sends per day is 10. QPS for sending emails = $\frac{10^9 \times 10}{10^5} = 100{,}000$.
- Assume the average number of emails a person receives in a day is 40 [3] and the average size of email metadata is 50 KB. Metadata refers to everything related to an email, excluding attachment files.
- Assume metadata is stored in a database. Storage requirement for maintaining metadata in 1 year: 1 billion users $\times$ 40 emails/day $\times$ 365 days $\times$ 50 KB = 730 PB.
- Assume 20% of emails contain an attachment and the average attachment size is 500 KB.
- Storage for attachments in 1 year is: 1 billion users $\times$40 emails/day $\times$ 365 days $\times$ 20% $\times$ 500 KB = 1,460 PB

From this back-of-the-envelope calculation, it's clear we would deal with a lot of data. So, it's likely that we need a distributed database solution.

Step 2 - Propose High-level Design and Get Buy-in

In this section, we first discuss some basics about email servers and how email servers evolve over time. Then we look at the high-level design of distributed email servers. The content is structured as follows:

- Email knowledge 101
- Traditional mail servers
- Distributed mail servers

Email knowledge 101

There are various email protocols that are used to send and receive emails. Historically, most mail servers use email protocols such as POP, IMAP, and SMTP.

Email protocols

SMTP: Simple Mail Transfer Protocol (SMTP) is the standard protocol for **sending** emails from one mail server to another.

The most popular protocols for **retrieving** emails are known as Post Office Protocol (POP) and the Internet Mail Access Protocol (IMAP).

POP is a standard mail protocol to receive and download emails from a remote mail server to a local email client. Once emails are downloaded to your computer or phone, they are deleted from the email server, which means you can only access emails on one computer or phone. The details of POP are covered in RFC 1939 [4]. POP requires mail clients to download the entire email. This can take a long time if an email contains a large attachment.

IMAP is also a standard mail protocol for receiving emails for a local email client. When you read an email, you are connected to an external mail server, and data is transferred to your local device. IMAP only downloads a message when you click it, and emails are not deleted from mail servers, meaning that you can access emails from multiple devices. IMAP is the most widely used protocol for individual email accounts. It works well when the connection is slow because only the email header information is downloaded until opened.

HTTPS is not technically a mail protocol, but it can be used to access your mailbox, particularly for web-based email. For example, it's common for Microsoft Outlook to talk to mobile devices over HTTPS, on a custom-made protocol called ActiveSync [5].

Domain name service (DNS)

A DNS server is used to look up the mail exchanger record (MX record) for the recipient's domain. If you run DNS lookup for gmail.com from the command line, you may get MX records as shown in Figure 8.2.

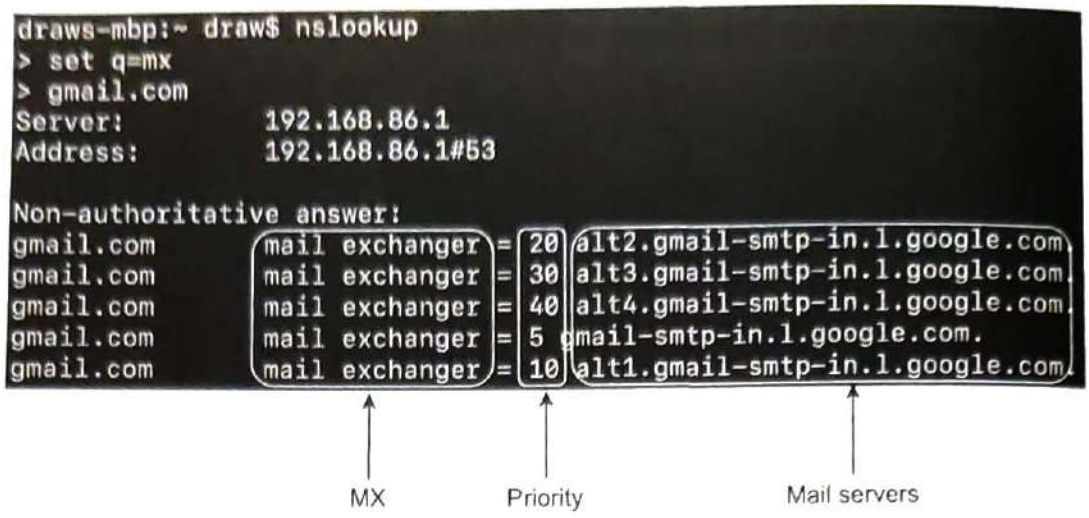


Figure 8.2: MX records

The priority numbers indicate preferences, where the mail server with a lower priority number is more preferred. In Figure 8.2, `gmail-smtp-in.l.google.com` is used first (priority 5). A sending mail server will attempt to connect and send messages to this mail server first. If the connection fails, the sending mail server will attempt to connect to the mail server with the next lowest priority, which is `alt1.gmail-smtp-in.l.google.com` (priority 10).

Attachment

An email attachment is sent along with an email message, commonly with Base64 encoding [6]. There is usually a size limit for an email attachment. For example, Outlook and Gmail limit the size of attachments to 20MB and 25MB respectively as of June 2021. This number is highly configurable and varies from individual to corporate accounts. Multipurpose Internet Mail Extension (MIME) [7] is a specification that allows the attachment to be sent over the internet.

Traditional mail servers

Before we dive into distributed mail servers, let's dig a little bit through the history and see how traditional mail servers work, as doing so provides good lessons about how to scale an email server system. You can consider a traditional mail server as a system that works when there are limited email users, usually on a single server.

Traditional mail server architecture

Figure 8.3 describes what happens when Alice sends an email to Bob, using traditional email servers.

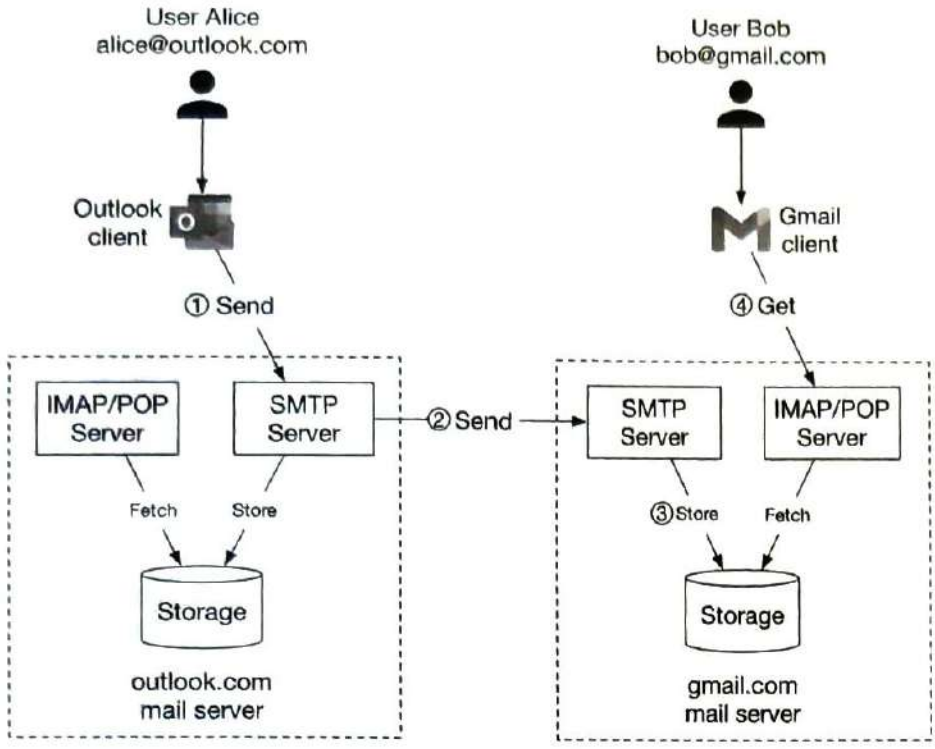


Figure 8.3: Traditional mail servers

The process consists of 4 steps:

1. Alice logs in to her Outlook client, composes an email, and presses the "send" button. The email is sent to the Outlook mail server. The communication protocol between the Outlook client and the mail server is SMTP.
2. Outlook mail server queries the DNS (not shown in the diagram) to find the address of the recipient's SMTP server. In this case, it is Gmail's SMTP server. Next, it transfers the email to the Gmail mail server. The communication protocol between the mail servers is SMTP.
3. The Gmail server stores the email and makes it available to Bob, the recipient.
4. Gmail client fetches new emails through the IMAP/POP server when Bob logs in to Gmail.

Storage

In a traditional mail server, emails were stored in local file directories and each email was stored in a separate file with a unique name. Each user maintained a user directory to store configuration data and mailboxes. Maildir was a popular way to store email messages on the mail server (Figure 8.4).

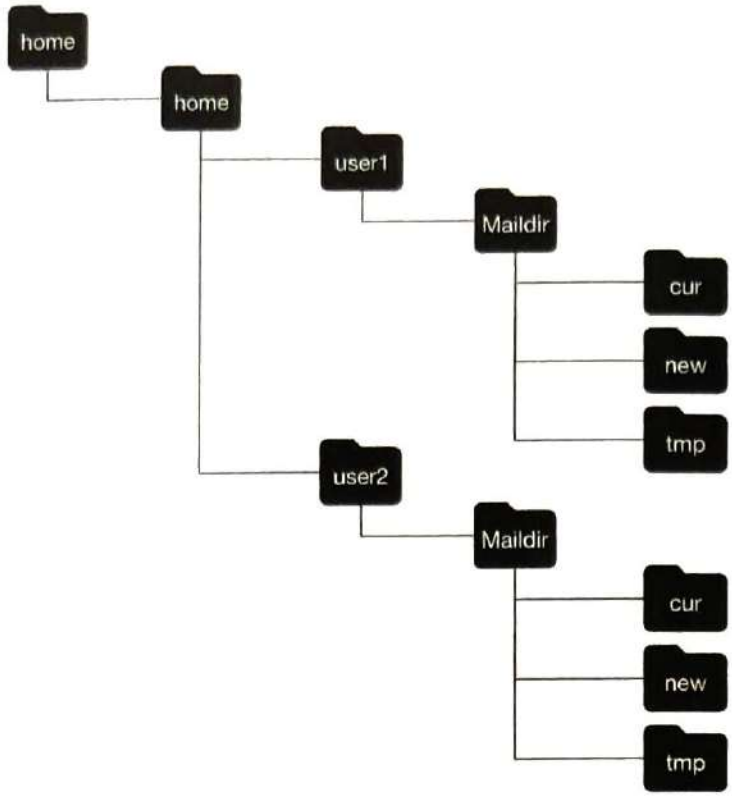


Figure 8.4: Maildir

File directories worked well when the user base was small, but it was challenging to retrieve and backup billions of emails. As the email volume grew and the file structure became more complex, disk I/O became a bottleneck. The local directories also don't satisfy our high availability and reliability requirements. The disk can be damaged and servers can go down. We need a more reliable distributed storage layer.

Email functionality has come a long way since it was invented in the 1960s, from text-based format to rich features such as multimedia, threading [8], search, labels, and more. But email protocols (POP, IMAP, and SMTP) were invented a long time ago and they were not designed to support these new features, nor were they scalable to support billions of users.

Distributed mail servers

Distributed mail servers are designed to support modern use cases and solve the problems of scale and resiliency. This section covers email APIs, distributed email server architecture, email sending, and email receiving flows.

Email APIs

Email APIs can mean very different things for different mail clients, or at different stages of an email's life cycle. For example;

- SMTP/POP/IMAP APIs for native mobile clients.
- SMTP communications between sender and receiver mail servers.

- RESTful API over HTTP for full-featured and interactive web-based email applications.

Due to the length limitations of this book, we cover only some of the most important APIs for webmail. A common way for webmail to communicate is through the HTTP protocol.

1. Endpoint: `POST /v1/messages`

Sends a message to the recipients in the To, Cc, and Bcc headers.

2. Endpoint: `GET /v1/folders`

Returns all folders of an email account.

Response:

```
[{id: string        Unique folder identifier.
  name: string      Name of the folder.
                    According to RFC6154 [9], the default folders can be one of
                    the following:
                    All, Archive, Drafts, Flagged, Junk, Sent, and Trash.
  user_id: string   Reference to the account owner
}]
```

3. Endpoint: `GET /v1/folders/{:folder_id}/messages`

Returns all messages under a folder. Keep in mind this is a highly simplified API. In reality, this needs to support pagination.

Response:

List of message objects.

4. Endpoint: `GET /v1/messages/{:message_id}`

Gets all information about a specific message. Messages are core building blocks for an email application, containing information about the sender, recipients, message subject, body, attachments, etc.

Response:

A message's object.

```
{
 user_id: string                  // Reference to the account owner.
 from: name: string, email: string // <name, email> pair of the sender.
 to: [name: string, email: string] // A list of <name, email> paris
 subject: string                  // Subject of an email
 body: string                     // Message body
 is_read: boolean                 // Indicate if a message is read or not.
}
```

Distributed mail server architecture

While it is easy to set up an email server that handles a small number of users, it is difficult to scale beyond one server. This is mainly because traditional email servers were designed to work with a single server only. Synchronizing data across servers can be difficult, and keeping emails from being misclassified as spam by recipients' mail servers is very challenging. In this section, we explore how to leverage cloud technologies to make it easier. The high-level design is shown in Figure 8.5.

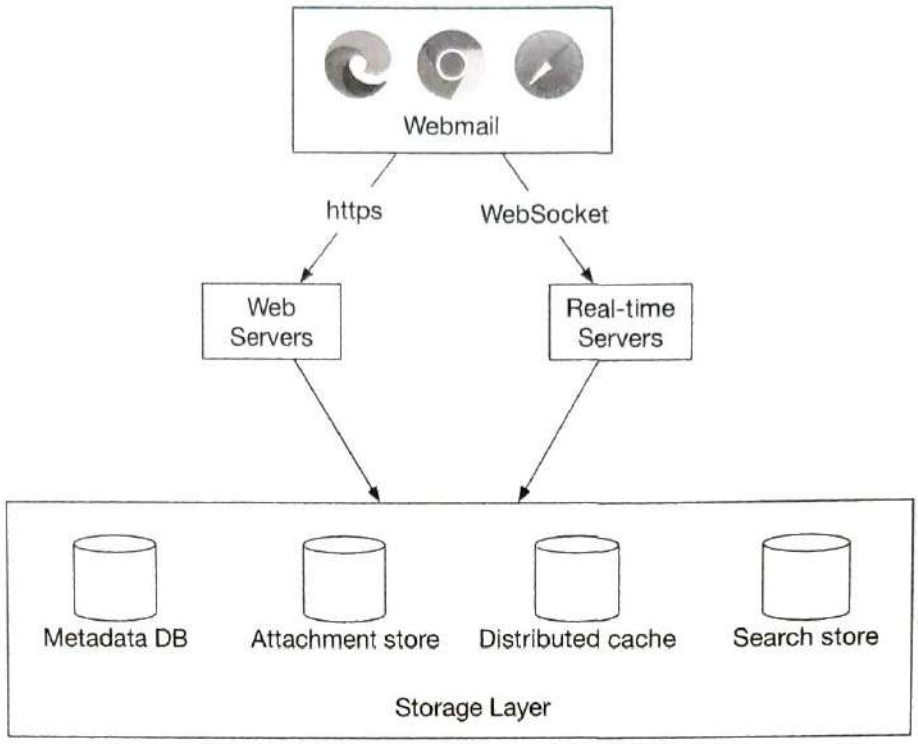


Figure 8.5: High-level design

Let us take a close look at each component.

Webmail. Users use web browsers to receive and send emails.

Web servers. Web servers are public-facing request/response services, used to manage features such as login, signup, user profile, etc. In our design, all email API requests, such as sending an email, loading mail folders, loading all mails in a folder, etc., go through web servers.

Real-time servers. Real-time servers are responsible for pushing new email updates to clients in real-time. Real-time servers are stateful servers because they need to maintain persistent connections. To support real-time communication, we have a few options, such as long polling and WebSocket. WebSocket is a more elegant solution but one drawback of it is browser compatibility. A possible solution is to establish a WebSocket connection whenever possible and to use long-polling as a fallback.

Here is an example of a real-world mail server (Apache James [10]) that implements the JSON Meta Application Protocol (JMAP) subprotocol over WebSocket [11].

Metadata database. This database stores mail metadata including mail subject, body,

from user, to users, etc. We discuss the database choice in the deep dive section.

Attachment store. We choose object stores such as Amazon Simple Storage Service (S3) as the attachment store. S3 is a scalable storage infrastructure that's suitable for storing large files such as images, videos, files, etc. Attachments can take up to 25 MB in size. NoSQL column-family databases like Cassandra might not be a good fit for the following two reasons:

- Even though Cassandra supports blob data type and its maximum theoretical size for a blob is 2GB, the practical limit is less than 1MB [12].
- Another problem with putting attachments in Cassandra is that we can't use a row cache as attachments take too much memory space.

Distributed cache. Since the most recent emails are repeatedly loaded by a client, caching recent emails in memory significantly improves the load time. We can use Redis here because it offers rich features such as lists and it is easy to scale.

Search store. The search store is a distributed document store. It uses a data structure called inverted index [13] that supports very fast full-text searches. We will discuss this in more detail in the deep dive section.

Now that we have discussed some of the most important components to build distributed mail servers, let's assemble together two main workflows.

- Email sending flow.
- Email receiving flow.

Email sending flow

The email sending flow is shown in Figure 8.6.

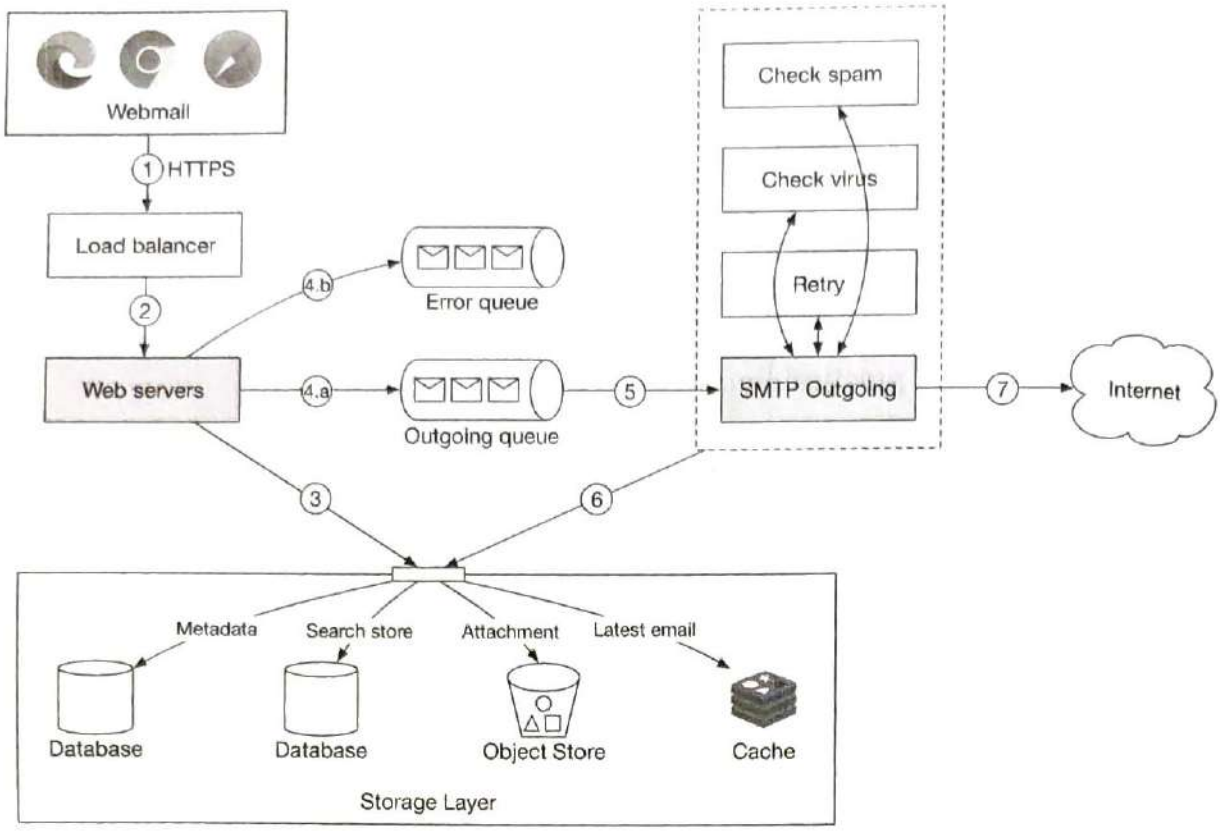


Figure 8.6: Email sending flow

1. A user writes an email on webmail and presses the send button. The request is sent to the load balancer.
2. The load balancer makes sure it doesn't exceed the rate limit and routes traffic to web servers.
3. Web servers are responsible for:
   - Basic email validation. Each incoming email is checked against pre-defined rules such as email size limit.
   - Checking if the domain of the recipient's email address is the same as the sender. If it is the same, the web server ensures the email data is spam and virus free. If so, email data is inserted into the sender's "Sent Folder" and recipient's "Inbox Folder". The recipient can fetch the email directly via the RESTful API. There is no need to go to step 4.
4. Message queues.
   4.1. If basic email validation succeeds, the email data is passed to the outgoing queue. If the attachment is too large to fit in the queue, we could store the attachment in the object store and save the object reference in the queued message.
   4.2. If basic email validation fails, the email is put in the error queue.
5. SMTP outgoing workers pull messages from the outgoing queue and make sure emails are spam and virus free.

6. The outgoing email is stored in the "Sent Folder" of the storage layer.
7. SMTP outgoing workers send the email to the recipient mail server.

Each message in the outgoing queue contains all the metadata required to create an email. A distributed message queue is a critical component that allows asynchronous mail processing. By decoupling SMTP outgoing workers from the web servers, we can scale SMTP outgoing workers independently.

We monitor the size of the outgoing queue very closely. If there are many emails stuck in the queue, we need to analyze the cause of the issue. Here are some possibilities:

- The recipient's mail server is unavailable. In this case, we need to retry sending the email at a later time. Exponential backoff [14] might be a good retry strategy.
- Not enough consumers to send emails. In this case, we may need more consumers to reduce the processing time.

Email receiving flow

The following diagram demonstrates the email receiving flow.

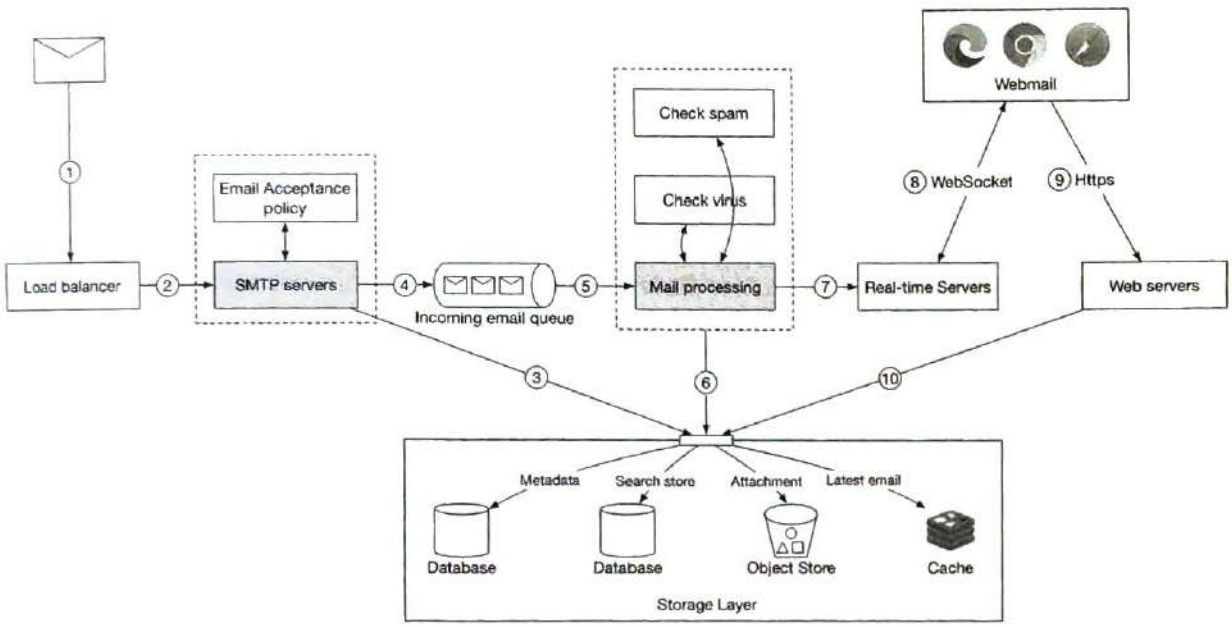


Figure 8.7: Email receiving flow

1. Incoming emails arrive at the SMTP load balancer.
2. The load balancer distributes traffic among SMTP servers. Email acceptance policy can be configured and applied at the SMTP-connection level. For example, invalid emails are bounced to avoid unnecessary email processing.
3. If the attachment of an email is too large to put into the queue, we can put it into the attachment store (S3).
4. Emails are put in the incoming email queue. The queue decouples mail processing

workers from SMTP servers so they can be scaled independently. Moreover, the queue serves as a buffer in case the email volume surges.

5. Mail processing workers are responsible for a lot of tasks, including filtering out spam mails, stopping viruses, etc. The following steps assume an email passed the validation.
6. The email is stored in the mail storage, cache, and object data store.
7. If the receiver is currently online, the email is pushed to real-time servers.
8. Real-time servers are WebSocket servers that allow clients to receive new emails in real-time.
9. For offline users, emails are stored in the storage layer. When a user comes back online, the webmail client connects to web servers via RESTful API.
10. Web servers pull new emails from the storage layer and return them to the client.

Step 3 - Design Deep Dive

Now that we have talked about all the parts of the email server, let's go deeper into some key components and examine how to scale the system.

- Metadata database
- Search
- Deliverability
- Scalability

Metadata database

In this section, we discuss the characteristics of email metadata, choosing the right database, data model, and conversation threads (bonus point).

Characteristics of email metadata

- Email headers are usually small and frequently accessed.
- Email body sizes can range from small to big but are infrequently accessed. You normally only read an email once.
- Most of the mail operations, such as fetching mails, marking an email as read, and searching are isolated to an individual user. In other words, mails owned by a user are only accessible by that user and all the mail operations are performed by the same user.
- Data recency impacts data usage. Users usually only read the most recent emails. 82% of read queries are for data younger than 16 days [15].
- Data has high-reliability requirements. Data loss is not acceptable.

Choosing the right database

At Gmail or Outlook scale, the database system is usually custom-made to reduce input/output operations per second (IOPS) [16], as this can easily become a major constraint in the system. Choosing the right database is not easy. It is helpful to consider all the options we have on the table before deciding the most suitable one.

- Relational database. The main motivation behind this is to search through emails efficiently. We can build indexes for email header and body. With indexes, simple search queries are fast. However, relational databases are typically optimized for small chunks of data entries and are not ideal for large ones. A typical email is usually larger than a few KB and can easily be over 100KB when HTML is involved. You might argue that the BLOB data type is designed to support large data entries. However, search queries over unstructured BLOB data type are not efficient. So relational databases such as MySQL or PostgreSQL are not good fits.
- Distributed object storage. Another potential solution is to store raw emails in cloud storage such as Amazon S3, which can be a good option for backup storage, but it's hard to efficiently support features such as marking emails as read, searching emails based on keywords, threading emails, etc.
- NoSQL databases. Google Bigtable is used by Gmail, so it's definitely a viable solution. However, Bigtable is not open sourced and how email search is implemented remains a mystery. Cassandra might be a good option as well, but we haven't seen any large email providers use it yet.

Based on the above analysis, very few existing solutions seem to fit our needs perfectly. Large email service providers tend to build their own highly customized databases. However, in an interview setting, we won't have time to design a new distributed database, though it's important to explain the following characteristics that the database should have.

- A single column can be a single-digit of MB.
- Strong data consistency.
- Designed to reduce disk I/O.
- It should be highly available and fault-tolerant.
- It should be easy to create incremental backups.

Data model

One way to store the data is to use `user_id` as a partition key so data for one user is stored on a single shard. One limitation of this data model is that messages are not shared among multiple users. Since this is not a requirement for us in this interview, it's not something we need to worry about.

Now let us define the tables. The primary key contains two components, the partition key, and the clustering key.

- Partition key: responsible for distributing data across nodes. As a general rule, we want to spread the data evenly.
- Clustering key: responsible for sorting data within a partition.

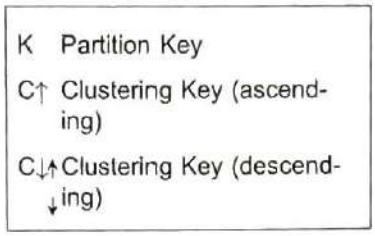
At a high level, an email service needs to support the following queries at the data layer:

- The first query is to get all folders for a user.
- The second query is to display all emails for a specific folder.
- The third query is to create/delete/get a specific email.
- The fourth query is to fetch all read or unread emails.
- Bonus point: get conversation threads.

Let's take a look at them one by one.

Query 1: get all folders for a user.

As shown in Table 8.1, `user_id` is the partition key, so folders owned by the same user are located in one partition.



folders_by_user		
user_id	UUID	K
folder_id	UUID	
folder_name	TEXT	

Table 8.1: Folders by user

Query 2: display all emails for a specific folder.

When a user loads their inbox, emails are usually sorted by timestamp, showing the most recent ones at the top. In order to store all emails for the same folder in one partition, composite partition key <`user_id`, `folder_id`> is used. Another column to note is `email_id`. Its data type is `TIMEUUID` [17], and it is the clustering key used to sort emails in chronological order.

emails_by_folder		
user_id	UUID	K
folder_id	UUID	K
email_id	TIMEUUID	C↓
from	TEXT	
subject	TEXT	
preview	TEXT	
is_read	BOOLEAN	

Table 8.2: Emails by folder

Query 3: create/delete/get an email

Due to space limitations, we only explain how to get detailed information about an email. The two tables in Table 8.3 are designed to support this query. The simple query looks like this:

```
SELECT * FROM emails_by_user WHERE email_id = 123;
```

An email can have multiple attachments, and these can be retrieved by the combination of `email_id` and `filename` fields.

emails_by_user		
user_id	UUID	K
email_id	TIMEUUID	C↓
from	TEXT	
to	LIST<TEXT>	
subject	TEXT	
body	TEXT	
attachments	LIST<filename|size>	

attachments		
email_id	TIMEUUID	C
filename	TEXT	K
url	TEXT	

Table 8.3: Emails by user

Query 4: fetch all read or unread emails

If our domain model was for a relational database, the query to fetch all read emails would look like this:

```
SELECT * FROM emails_by_folder
WHERE user_id = <user_id> and folder_id = <folder_id> and
  is_read = true
ORDER BY email_id;
```

The query to fetch all unread emails would look very similar. We just need to change `is_read = true` to `is_read = false` in the above query.

Our data model, however, is designed for NoSQL. A NoSQL database normally only supports queries on partition and cluster keys. Since `is_read` in the `emails_by_folder` table is neither of those, most NoSQL databases will reject this query.

One way to get around this limitation is to fetch the entire folder for a user and perform the filtering in the application. This could work for a small email service, but at our design scale, this does not work well.

This problem is commonly solved with denormalization in NoSQL. To support the read-/unread queries, we denormalize the `emails_by_folder` data into two tables as shown in Table 8.4.

- `read_emails`: it stores all emails that are in read status.
- `unread_emails`: it stores all emails that are in unread status.

To mark an `UNREAD` email as `READ`, the email is deleted from `unread_emails` and then inserted to `read_emails`.

To fetch all unread emails for a specific folder, we can run a query like this:

```
SELECT * FROM unread_emails
WHERE user_id = <user_id> and folder_id = <folder_id>
ORDER BY email_id;
```

read_emails		
user_id	UUID	K
folder_id	UUID	K
email_id	TIMEUUID	C↓
from	TEXT	
subject	TEXT	
preview	TEXT	

unread_emails		
user_id	UUID	K
folder_id	UUID	K
email_id	TIMEUUID	C↓
from	TEXT	
subject	TEXT	
preview	TEXT	

Table 8.4: Read and unread emails

Denormalization as shown above is a common practice. It makes the application code more complicated and harder to maintain, but it improves the read performance of these queries at scale.

Bonus point: conversation threads

Threads are a feature supported by many email clients. It groups email replies with their original message [8]. This allows users to retrieve all emails associated with one conversation. Traditionally, a thread is implemented using algorithms such as JWZ algorithm [18]. We will not go into detail about the algorithm, but just explain the core idea behind it. An email header generally contains the following three fields:

```
{
  "headers" {
    "Message-Id": "<7BA04B2A-430C-4D12-8B57-862103C34501@gmail.
        com>",
    "In-Reply-To": "<CAEWTXuPfN=LzECjDJtgY9Vu03kgFvJnJUSHTt6
        TW@gmail.com>",
    "References": ["<7BA04B2A-430C-4D12-8B57-862103C34501@gmail
        .com>"]
  }
}
```

Message-Id	The value of a message ID. It is generated by a client while sending a message.
In-Reply-To	The parent Message-Id to which the message replies.
References	A list of message IDs related to a thread.

Table 8.5: Email header

With these fields, an email client can reconstruct mail conversations from messages, if all messages in the reply chain are preloaded.

Consistency trade-off

Distributed databases that rely on replication for high availability must make a fundamental trade-off between consistency and availability. Correctness is very important for email systems, so by design, we want to have a single primary for any given mailbox. In the event of a failover, the mailbox isn't accessible by clients, so their sync/update operation is paused until failover ends. It trades availability in favor of consistency.

Email deliverability

It is easy to set up a mail server and start sending emails. The hard part is to get emails actually delivered to a user's inbox. If an email ends up in the spam folder, it means there is a very high chance a recipient won't read it. Email spam is a huge issue. According to research done by Statista [19], more than 50% of all emails sent are spam. If we set up a new mail server, most likely our emails will end up in the spam folder because a new email server has no reputation. There are a couple of factors to consider to improve email deliverability.

Dedicated IPs. It is recommended to have dedicated IP addresses for sending emails. Email providers are less likely to accept emails from new IP addresses that have no history.

Classify emails. Send different categories of emails from different IP addresses. For example, you may want to avoid sending marketing and other important emails from the same servers because it might make ISPs mark all emails as promotional.

Email sender reputation. Warm up new email server IP addresses slowly to build a good reputation, so big providers such as Office365, Gmail, Yahoo Mail, etc. are less likely to put our emails in the spam folder. According to Amazon Simple Email Service [20], it takes about 2 to 6 weeks to warm up a new IP address.

Ban spammers quickly. Spammers should be banned quickly before they have a significant impact on the server's reputation.

Feedback processing. It's very important to set up feedback loops with ISPs so we can keep the complaint rate low and ban spam accounts quickly. If an email fails to deliver or a user complains, one of the following outcomes occurs:

- Hard bounce. This means an email is rejected by ISP because the recipient's email address is invalid.
- Soft bounce. A soft bounce indicates an email failed to deliver due to temporary conditions, such as ISPs being too busy.
- Complaint. This means a recipient clicks the "report spam" button.

Figure 8.8 shows the process of collecting and processing bounces/complaints. We use separate queues for soft bounces, hard bounces, and complaints so they can be managed separately.

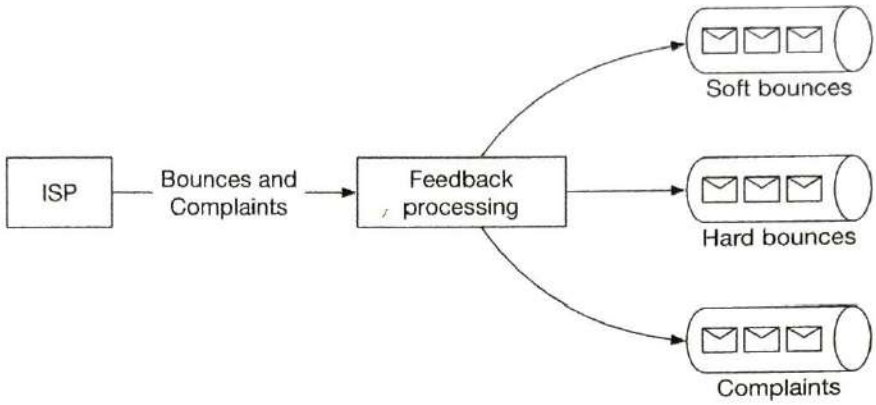


Figure 8.8: Handle feedback loop

Email authentication. According to the 2018 data breach investigation report provided by Verizon, phishing and pretexting represent 93% of breaches [21]. Some of the common techniques to combat phishing are: Sender Policy Framework (SPF) [22], DomainKeys Identified Mail (DKIM) [23], and Domain-based Message Authentication, Reporting and Conformance (DMARC) [24].

Figure 8.9 shows an example header of a Gmail message. As you can see, the sender `@info6.citi.com` is authenticated by SPF, DKIM, and DMARC.

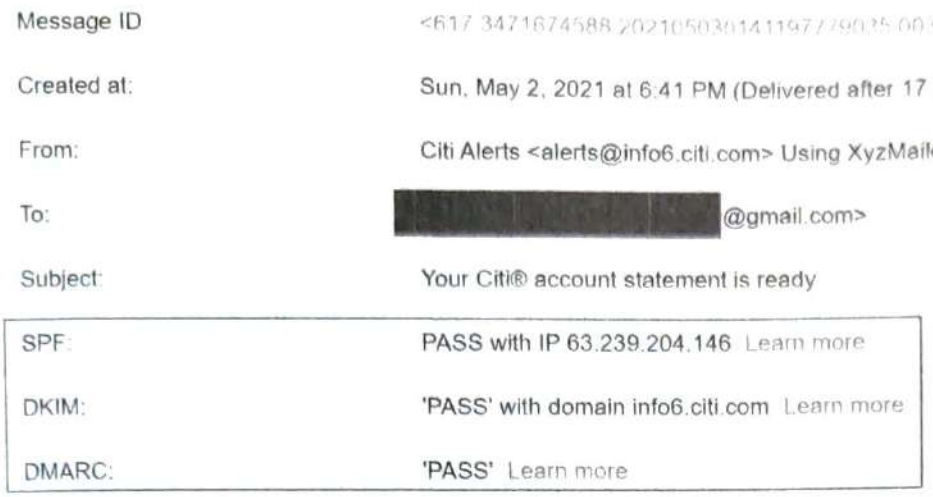


Figure 8.9: An example of a Gmail header

You don't need to remember all those terms. The important thing to keep in mind is that getting emails to work as intended is hard. It requires not only domain knowledge, but good relationships with ISPs.

Search

Basic mail search refers to searching for emails that contain any of the entered keywords in the subject or body. More advanced features include filtering by "From", "Subject", "Unread", or other attributes. On one hand, whenever an email is sent, received, or deleted, we need to perform reindexing. On the other hand, a search query is only run when a user presses the search button. This means the search feature in email systems has a lot more writes than reads. By comparison with Google search, email search has quite different characteristics, as shown in Table 8.6.

	Scope	**Sorting**	**Accuracy**
Google search	The whole internet	Sort by relevance	Indexing generally takes time, so some items may not show in the search result immediately.
Email search	User's own email box	Sort by attributes such as time, has attachment, date within, is unread, etc.	Indexing should be near real-time, and the result has to be accurate.

Table 8.6: Google search vs email search

To support search functionality, we compare two approaches: Elasticsearch and native search embedded in the datastore.

Option 1: Elasticsearch

The high-level design for email search using Elasticsearch is shown in Figure 8.10. Because queries are mostly performed on the user's own email server, we can group underlying documents to the same node using `user_id` as the partition key.

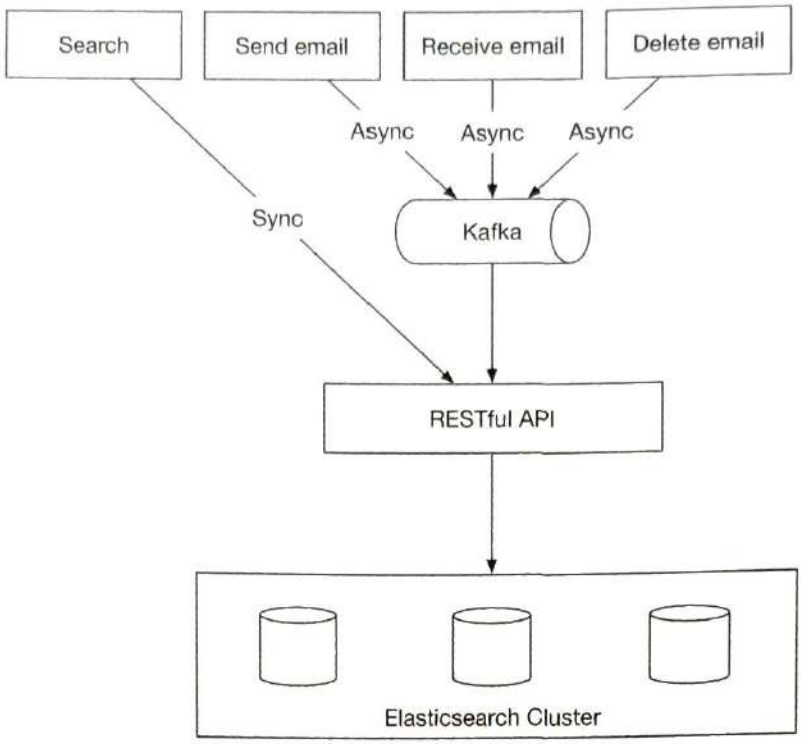


Figure 8.10: Elasticsearch

When a user clicks the search button, the user waits until the search response is received. A search request is synchronous. When events such as "send email", "receive email" or "delete email" are triggered, nothing related to search needs to be returned to the client. Reindexing is needed and it can be done with offline jobs. Kafka is used in the design to decouple services that trigger reindexing, from services that actually perform reindexing.

Elasticsearch is the most popular search-engine database as of June 2021 [25] and it supports full-text search of emails very well. One challenge of adding Elasticsearch is to keep our primary email store in sync with it.

Option 2: Custom search solution

Large-scale email providers usually develop their own custom search engines to meet their specific requirements. Designing an email search engine is a very complicated task and is out of the scope of this chapter. Here we only briefly touch on the disk I/O bottleneck, a primary challenge we will face for a custom search engine.

As shown in the back-of-the-envelope calculation, the size of the metadata and attachments added daily is at the petabyte (PB) level. Meanwhile, an email account can easily have over half a million emails. The main bottleneck of the index server is usually disk I/O.

Since the process of building the index is write-heavy, a good strategy might be to use Log-Structured Merge-Tree (LSM) [26] to structure the index data on disk (Figure 8.11). The write path is optimized by only performing sequential writes. LSM trees are the core data structure behind databases such as Bigtable, Cassandra, and RocksDB. When a new email arrives, it is first added to level 0 in-memory cache, and when data size in memory reaches the predefined threshold, data is merged to the next level. Another reason to use LSM is to separate data that change frequently from those that don't. For example, email data usually doesn't change, but folder information tends to change more often due to different filter rules. In this case, we can separate them into two different sections, so that if a request is related to a folder change, we change only the folder and leave the email data alone.

If you are interested in reading more about email search, it is highly recommended you take a look at how search works in Microsoft Exchange servers [27].

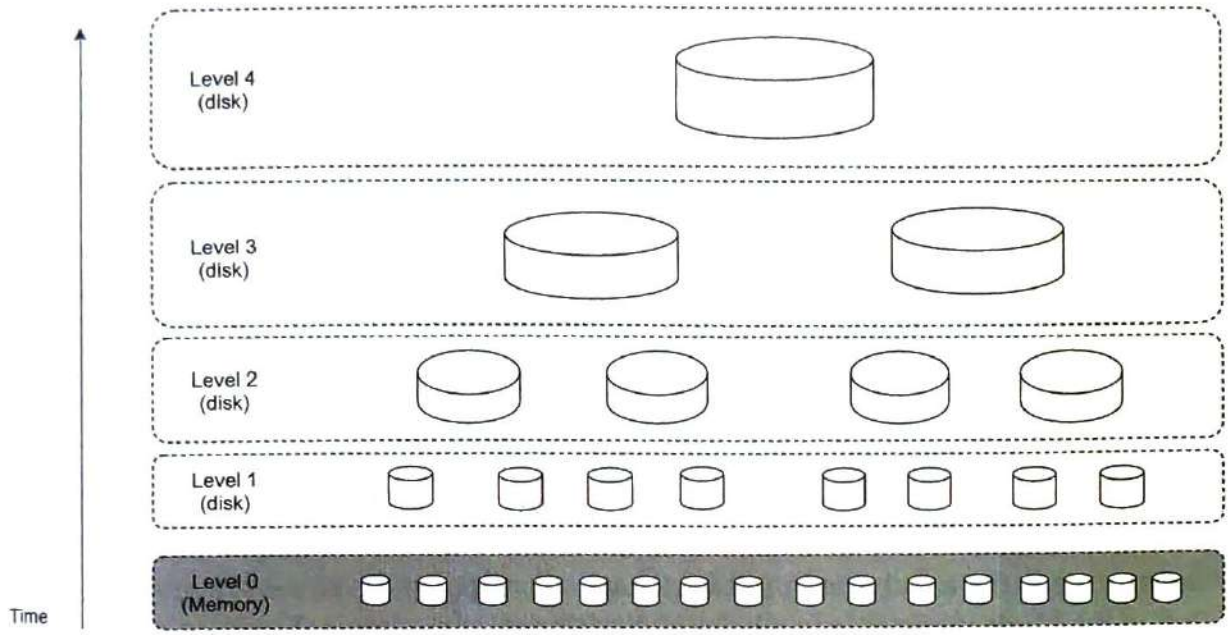


Figure 8.11: LSM tree

Each approach has pros and cons:

Feature	Elasticsearch	Custom search engine
Scalability	Scalable to some extent	Easier to scale as we can optimize the system for the email use case
System complexity	Need to maintain two different systems: datastore and Elasticsearch	One system
Data consistency	Two copies of data. One in the metadata datastore, and the other in Elasticsearch. Data consistency is hard to maintain	A single copy of data in the metadata datastore
Data loss possible	No. Can rebuild the Elasticsearch index from the primary storage, in case of failure	No
Development effort	Easy to integrate. To support large scale email search, a dedicated Elasticsearch team might be needed	Significant engineering effort is needed to develop a custom email search engine

Table 8.7: Elastic search vs custom search engine

A general rule of thumb is that for a smaller scale email system, Elasticsearch is a good option as it's easy to integrate and doesn't require significant engineering effort. For a larger scale, Elasticsearch might work, but we may need a dedicated team to develop and maintain the email search infrastructure. To support an email system at Gmail or Outlook scale, it might be a good idea to have a native search embedded in the database as opposed to the separate indexing approach.

Scalability and availability

Since data access patterns of individual users are independent of one another, we expect most components in the system are horizontally scalable.

For better availability, data is replicated across multiple data centers. Users communicate with a mail server that is physically closer to them in the network topology. During a network partition, users can access messages from other data centers (Figure 8.12).

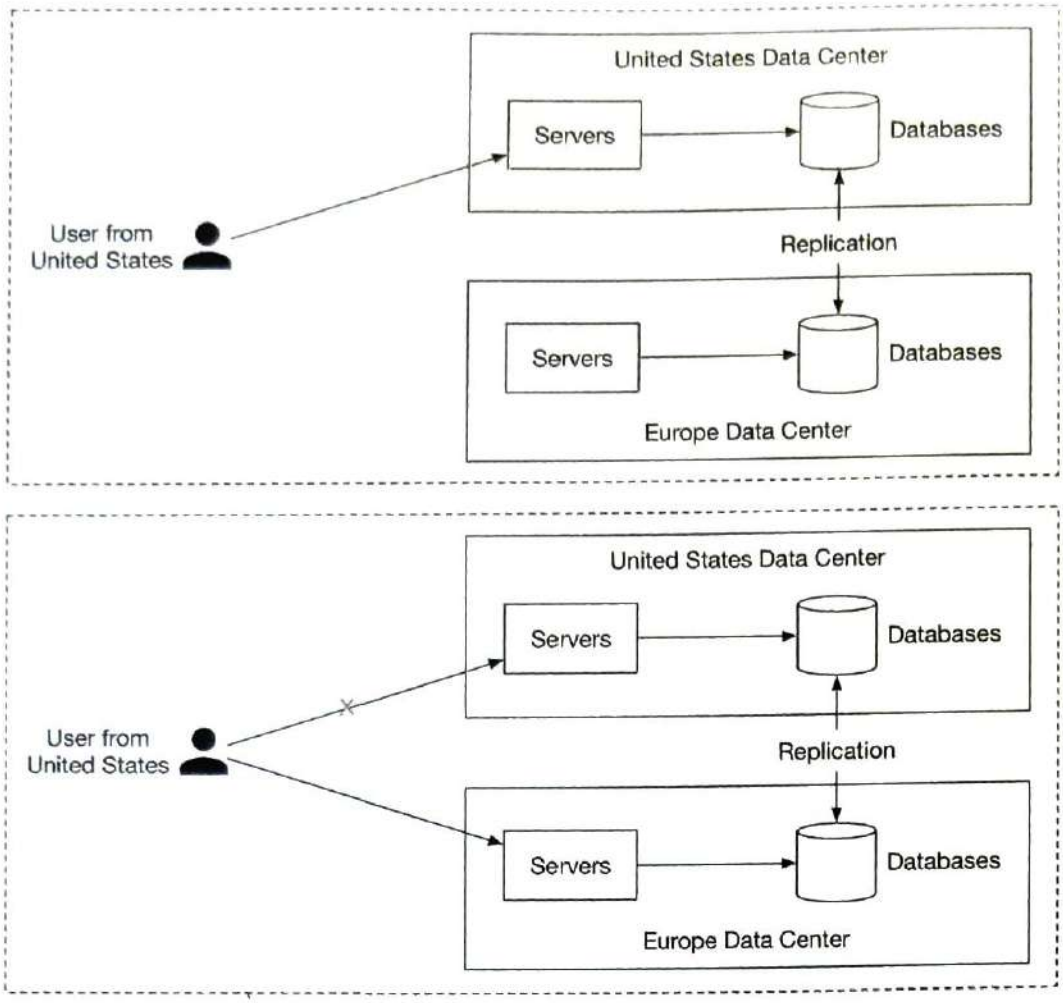


Figure 8.12: Multi-data center setup

Step 4 - Wrap Up

In this chapter, we have presented a design for building large-scale email servers. We started by gathering requirements and doing some back-of-the-envelope calculations to get a good idea of the scale. In the high-level design, we discussed how traditional email servers were designed and why they cannot satisfy modern use cases. We also discussed email APIs and high-level designs for sending and receiving flows. Finally, we dived deep into metadata database design, email deliverability, search, and scalability.

If there is extra time at the end of the interview, here are a few additional talking points:

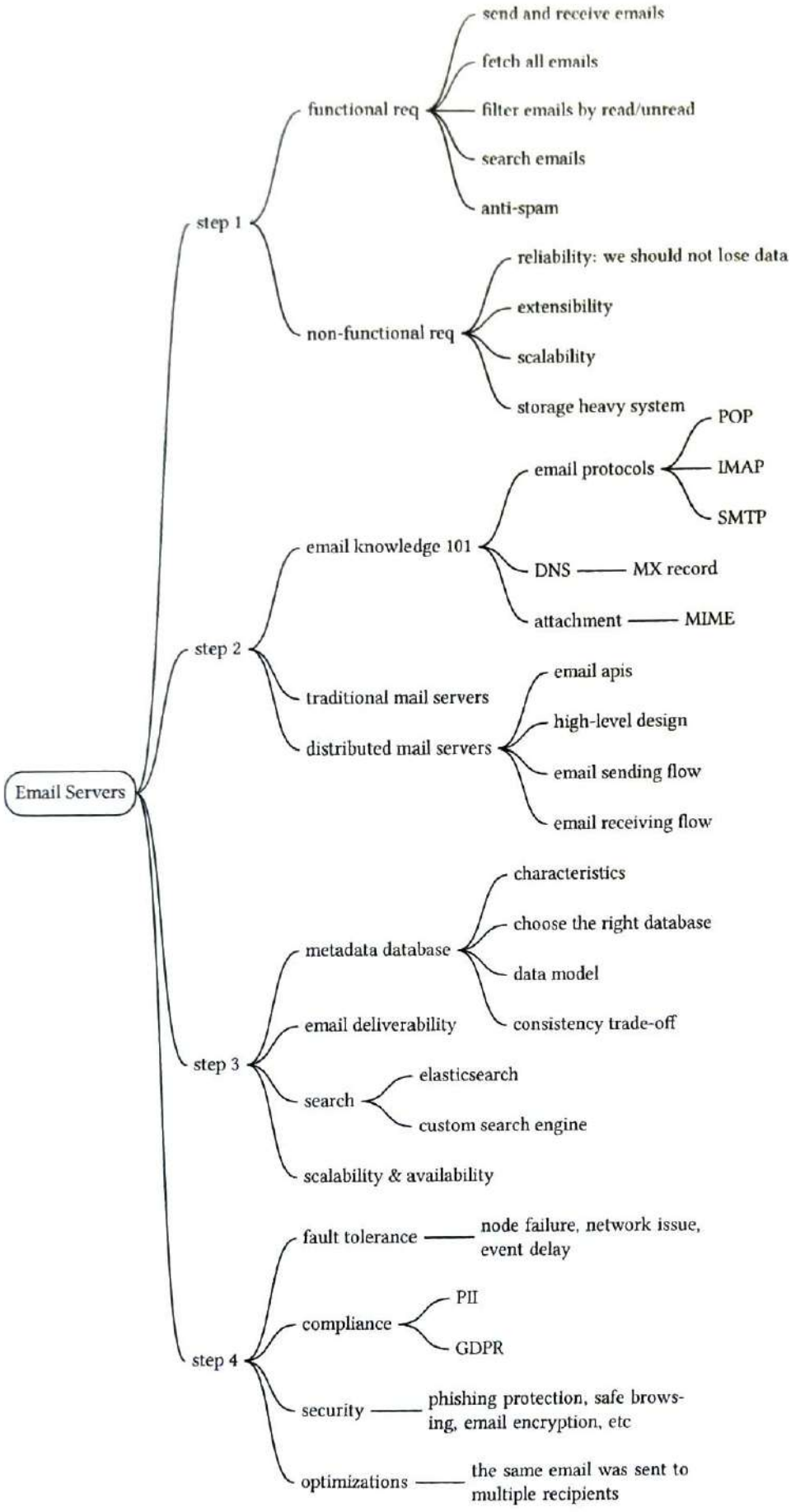
- Fault tolerance. Many parts of the system can fail, and you can talk about how to handle node failures, network issues, event delays, etc.
- Compliance. Email service works all around the world and there are legal regulations to comply with. For instance, we need to handle and store personally identifiable information (PII) from Europe in a way that complies with General Data Protection Regulation (GDPR) [28]. Legal intercept is another typical feature in this area [29].
- Security. Email security is important because emails contain sensitive information.

Gmail provides safety features such as phishing protections, safe browsing, proactive alerts, account safety, confidential mode, and email encryption [30].

- Optimizations. Sometimes, the same email is sent to multiple recipients, and the same email attachment is stored several times in the object store (S3) in the group emails. One optimization we could do is to check the existence of the attachment in storage, before performing the expensive save operation.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Chapter Summary



Reference Material

[1] Number of Active Gmail Users. https://financesonline.com/number-of-active-gmail-users/.

[2] Outlook. https://en.wikipedia.org/wiki/Outlook.com.

[3] How Many Emails Are Sent Per Day in 2021? https://review42.com/resources/how-many-emails-are-sent-per-day/.

[4] RFC 1939 - Post Office Protocol - Version 3. http://www.faqs.org/rfcs/rfc1939.html.

[5] ActiveSync. https://en.wikipedia.org/wiki/ActiveSync.

[6] Email attachment. https://en.wikipedia.org/wiki/Email_attachment.

[7] MIME. https://en.wikipedia.org/wiki/MIME.

[8] Threading. https://en.wikipedia.org/wiki/Conversation_threading.

[9] IMAP LIST Extension for Special-Use Mailboxes. https://datatracker.ietf.org/doc/html/rfc6154.

[10] Apache James. https://james.apache.org/.

[11] A JSON Meta Application Protocol (JMAP) Subprotocol for WebSocket. https://tools.ietf.org/id/draft-ietf-jmap-websocket-07.html#RFC7692.

[12] Cassandra Limitations. https://cwiki.apache.org/confluence/display/CASSANDRA2/CassandraLimitations.

[13] Inverted index. https://en.wikipedia.org/wiki/Inverted_index.

[14] Exponential backoff. https://en.wikipedia.org/wiki/Exponential_backoff.

[15] QQ Email System Optimization (in Chinese). https://www.slideshare.net/areyouok/06-qq-5431919.

[16] IOPS. https://en.wikipedia.org/wiki/IOPS.

[17] UUID and timeuuid types. https://docs.datastax.com/en/cql-oss/3.3/cql/cql_reference/uuid_type_r.html.

[18] Message threading. https://www.jwz.org/doc/threading.html.

[19] Global spam volume. https://www.statista.com/statistics/420391/spam-email-traffic-share/.

[20] Warming up dedicated IP addresses. https://docs.aws.amazon.com/ses/latest/dg/dedicated-ip-warming.html.

[21] 2018 Data Breach Investigations Report. https://enterprise.verizon.com/resources/reports/DBIR_2018_Report.pdf.

[22] Sender Policy Framework. https://en.wikipedia.org/wiki/Sender_Policy_Framework.

[23] DomainKeys Identified Mail. https://en.wikipedia.org/wiki/DomainKeys_Identified_Mail.

[24] Domain-based Message Authentication, Reporting & Conformance. https://dmarc.org/.

[25] DB-Engines Ranking of Search Engines. https://db-engines.com/en/ranking/search+engine.

[26] Log-structured merge-tree. https://en.wikipedia.org/wiki/Log-structured_merge-tree.

[27] Microsoft Exchange Conference 2014 Search in Exchange. https://www.youtube.com/watch?v=5EXGCSzzQak&t=2173s.

[28] General Data Protection Regulation. https://en.wikipedia.org/wiki/General_Data_Protection_Regulation.

[29] Lawful interception. https://en.wikipedia.org/wiki/Lawful_interception.

[30] Email safety. https://safety.google/intl/en_us/gmail/.

9 S3-like Object Storage

In this chapter, we design an object storage service similar to Amazon Simple Storage Service (S3). S3 is a service offered by Amazon Web Services (AWS) that provides object storage through a RESTful API-based interface. Here are some facts about AWS S3:

- Launched in June 2006.
- S3 added versioning, bucket policy, and multipart upload support in 2010.
- S3 added server-side encryption, multi-object delete, and object expiration in 2011.
- Amazon reported 2 trillion objects stored in S3 by 2013.
- Life cycle policy, event notification, and cross-region replication support were introduced in 2014 and 2015.
- Amazon reported over 100 trillion objects stored in S3 by 2021.

Before we dig into object storage, let's first review storage systems in general and define some terminologies.

Storage System 101

At a high-level, storage systems fall into three broad categories:

- Block storage
- File storage
- Object storage

Block storage

Block storage came first, in the 1960s. Common storage devices like hard disk drives (HDD) and solid-state drives (SSD) that are physically attached to servers are all considered as block storage.

Block storage presents the raw blocks to the server as a volume. This is the most flexible and versatile form of storage. The server can format the raw blocks and use them as a file system, or it can hand control of those blocks to an application. Some applications like

a database or a virtual machine engine manage these blocks directly in order to squeeze every drop of performance out of them.

Block storage is not limited to physically attached storage. Block storage could be connected to a server over a high-speed network or over industry-standard connectivity protocols like Fibre Channel (FC) [1] and iSCSI [2]. Conceptually, the network-attached block storage still presents raw blocks. To the servers, it works the same as physically attached block storage.

File storage

File storage is built on top of block storage. It provides a higher-level abstraction to make it easier to handle files and directories. Data is stored as files under a hierarchical directory structure. File storage is the most common general-purpose storage solution. File storage could be made accessible by a large number of servers using common file-level network protocols like SMB/CIFS [3] and NFS [4]. The servers accessing file storage do not need to deal with the complexity of managing the blocks, formatting volume, etc. The simplicity of file storage makes it a great solution for sharing a large number of files and folders within an organization.

Object storage

Object storage is new. It makes a very deliberate tradeoff to sacrifice performance for high durability, vast scale, and low cost. It targets relatively "cold" data and is mainly used for archival and backup. Object storage stores all data as objects in a flat structure. There is no hierarchical directory structure. Data access is normally provided via a RESTful API. It is relatively slow compared to other storage types. Most public cloud service providers have an object storage offering, such as AWS S3, Google object storage, and Azure blob storage.

Comparison

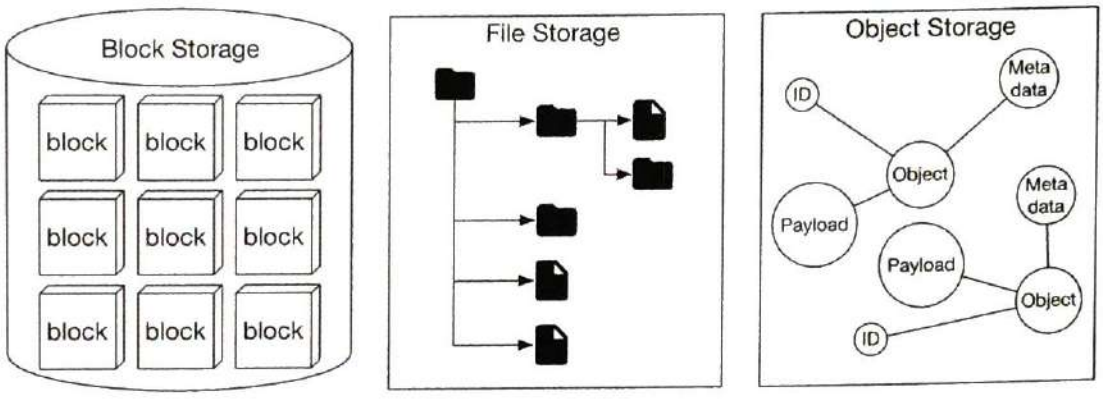


Figure 9.1: Three different storage options

Table 9.1 compares block storage, file storage, and object storage.

	Block storage	File storage	Object storage
Mutable Content	Y	Y	N (object versioning is supported, in-place update is not)
Cost	High	Medium to high	Low
Performance	Medium to high, very high	Medium to high	Low to medium
Consistency	Strong consistency	Strong consistency	Strong consistency [5]
Data access	SAS [6]/iSCSI/FC	Standard file access, CIFS/SMB, and NFS	RESTful API
Scalability	Medium scalability	High scalability	Vast scalability
Good for	Virtual machines (VM), high-performance applications like database	General-purpose file system access	Binary data, unstructured data

Table 9.1: Storage options

Terminology

To design S3-like object storage, we need to understand some core object storage concepts first. This section provides an overview of the terms that apply to object storage.

Bucket. A logical container for objects. The bucket name is globally unique. To upload data to S3, we must first create a bucket.

Object. An object is an individual piece of data we store in a bucket. It contains object data (also called payload) and metadata. Object data can be any sequence of bytes we want to store. The metadata is a set of name-value pairs that describe the object.

Versioning. A feature that keeps multiple variants of an object in the same bucket. It is enabled at bucket-level. This feature enables users to recover objects that are deleted or overwritten by accident.

Uniform Resource Identifier (URI). The object storage provides RESTful APIs to access its resources, namely, buckets and objects. Each resource is uniquely identified by its URI.

Service-level agreement (SLA). A service-level agreement is a contract between a service provider and a client. For example, the Amazon S3 Standard-Infrequent Access storage class provides the following SLA [7]:

- Designed for durability of 99.999999999% of objects across multiple Availability Zones.
- Data is resilient in the event of one entire Availability Zone destruction.

- Designed for 99.9% availability.

Step 1 - Understand the Problem and Establish Design Scope

The following questions help to clarify the requirements and narrow down the scope.

Candidate: Which features should be included in the design?
Interviewer: We would like you to design an S3-like object storage system with the following functionalities:

- Bucket creation.
- Object uploading and downloading.
- Object versioning.
- Listing objects in a bucket. It's similar to the `aws S3 ls` command [8].

Candidate: What is the typical data size?
Interviewer: We need to store both massive objects (a few GBs or more) and a large number of small objects (tens of KBs,) efficiently.

Candidate: How much data do we need to store in one year?
Interviewer: 100 petabytes (PB).

Candidate: Can we assume data durability is 6 nines (99.9999%) and service availability is 4 nines (99.99%)?
Interviewer: Yes, that sounds reasonable.

Non-functional requirements

- 100PB of data
- Data durability is 6 nines
- Service availability is 4 nines
- Storage efficiency. Reduce storage costs while maintaining a high degree of reliability and performance.

Back-of-the-envelope estimation

Object storage is likely to have bottlenecks in either disk capacity or disk IO per second (IOPS). Let's take a look.

- Disk capacity. Let's assume objects follow the distribution listed below:
  - 20% of all objects are small objects (less than 1MB).
  - 60% of objects are medium-sized objects (1 MB $\sim$ 64MB).
  - 20% are large objects (larger than 64MB).
- IOPS. Let's assume one hard disk (SATA interface, 7200 rpm) is capable of doing

$100 \sim 150$ random seeks per second ($100 \sim 150$ IOPS).

With those assumptions, we can estimate the total number of objects the system can persist. To simplify the calculation, let's use the median size for each object type (0.5MB for small objects, 32MB for medium objects, and 200MB for large objects). A 40% storage usage ratio gives us:

- $100\,\text{PB} = 100 \times 1000 \times 1000 \times 1000\,\text{MB} = 10^{11}\,\text{MB}$
- $\frac{10^{11} \times 0.4}{(0.2 \times 0.5\,\text{MB} + 0.6 \times 32\,\text{MB} + 0.2 \times 200\,\text{MB})} = 0.68$ billion objects.
- If we assume the metadata of an object is about 1KB in size, we need 0.68TB space to store all metadata information.

Even though we may not use those numbers, it's good to have a general idea about the scale and constraint of the system.

Step 2 - Propose High-level Design and Get Buy-in

Before diving into the design, let's explore a few interesting properties of object storage, as they may influence it.

Object immutability. One of the main differences between object storage and the other two types of storage systems is that the objects stored inside of object storage are immutable. We may delete them or replace them entirely with a new version, but we cannot make incremental changes.

Key-value store. We could use object URI to retrieve object data (Listing 9.1). The object URI is the key and object data is the value.

```
Request:
GET /bucket1/object1.txt HTTP/1.1

Response:
HTTP/1.1 200 OK
Content-Length: 4567

[4567 bytes of object data]
```

Listing 9.1: Use object URI to retrieve object data

Write once, read many times. The data access pattern for object data is written once and read many times. According to the research done by LinkedIn, 95% of requests are read operations [9].

Support both small and large objects. Object size may vary and we need to support both.

The design philosophy of object storage is very similar to that of the UNIX file system. In UNIX, when we save a file in the local file system, it does not save the filename and file data together. Instead, the filename is stored in a data structure called "inode" [10],

and the file data is stored in different disk locations. The inode contains a list of file block pointers that point to the disk locations of the file data. When we access a local file, we first fetch the metadata in the inode. We then read the file data by following the file block pointers to the actual disk locations.

The object storage works similarly. The inode becomes the metadata store that stores all the object metadata. The hard disk becomes the data store that stores the object data. In the UNIX file system, the inode uses the file block pointer to record the location of data on the hard disk. In object storage, the metadata store uses the ID of the object to find the corresponding object data in the data store, via a network request. Figure 9.2 shows the UNIX file system and the object storage.

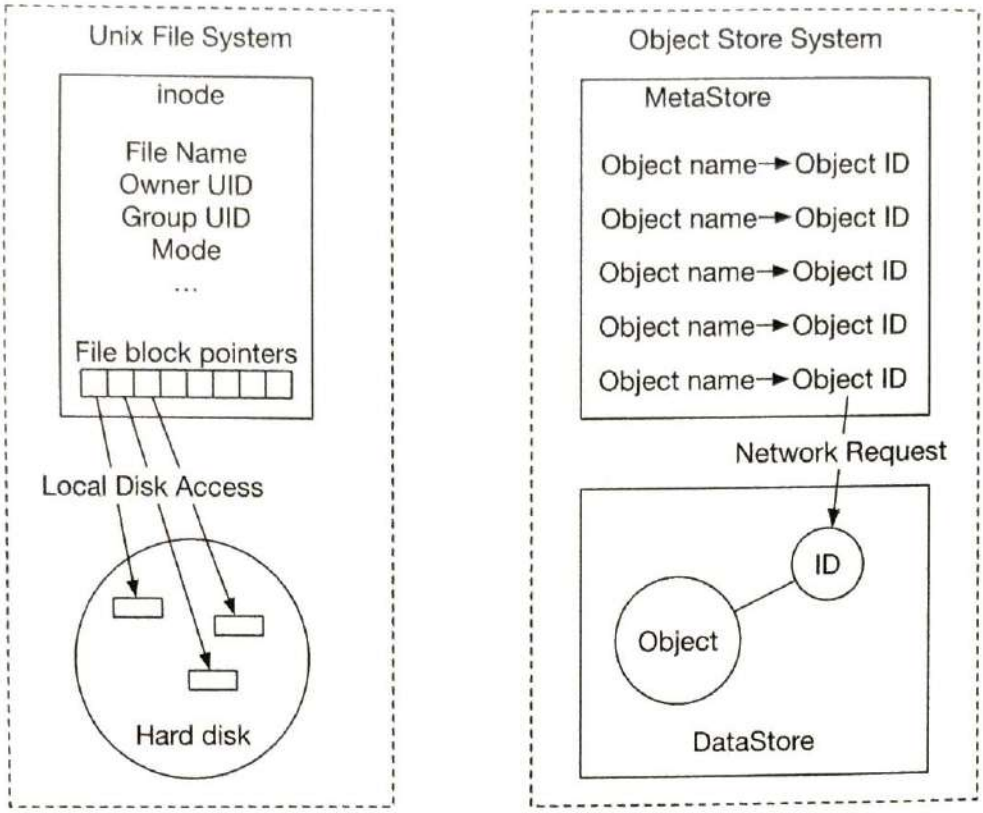


Figure 9.2: UNIX file system and object store

Separating metadata and object data simplifies the design. The data store contains immutable data while the metadata store contains mutable data. This separation enables us to implement and optimize these two components independently. Figure 9.3 shows what the bucket and object look like.

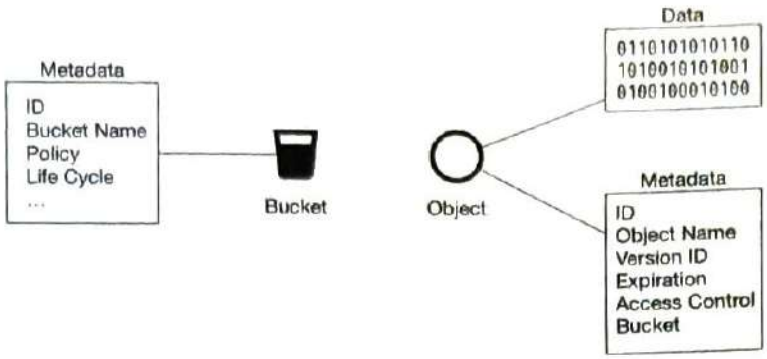


Figure 9.3: Bucket & object

High-level design

Figure 9.4 shows the high-level design.

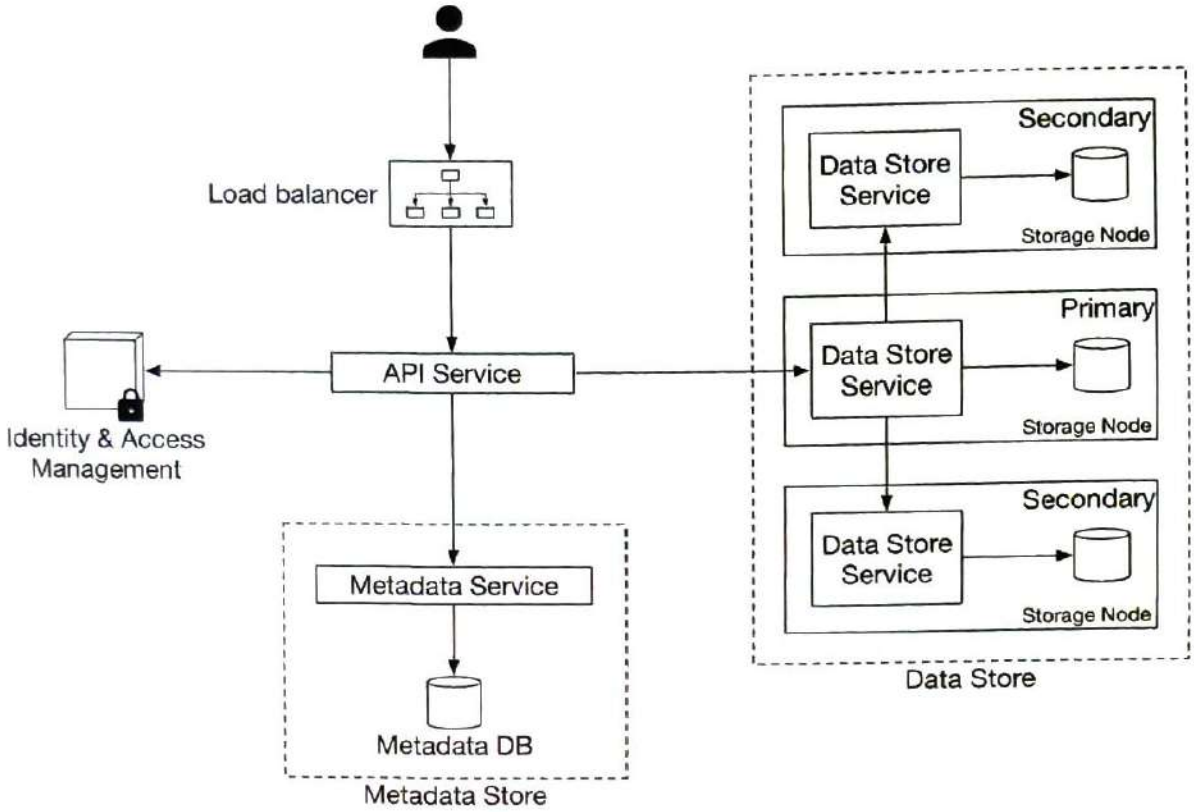


Figure 9.4: High-level design

Let's go over the components one by one.

Load balancer. Distributes RESTful API requests across a number of API servers.

API service. Orchestrates remote procedure calls to the identity and access management service, metadata service, and storage stores. This service is stateless so it can be horizontally scaled.

Identity and access management (IAM). The central place to handle authentication, authorization, and access control. Authentication verifies who you are, and authorization validates what operations you could perform based on who you are.

Data store. Stores and retrieves the actual data. All data-related operations are based on object ID (UUID).

Metadata store. Stores the metadata of the objects.

Note that the metadata and data stores are just logical components, and there are different ways to implement them. For example, in Ceph's Rados Gateway [11], there is no standalone metadata store. Everything, including the object bucket, is persisted as one or multiple Rados objects.

Now we have a basic understanding of the high-level design, let's explore some of the most important workflows in object storage.

- Uploading an object.
- Downloading an object.
- Object versioning and listing objects in a bucket. They will be explained in the "design deep dive" section on page 263.

Uploading an object

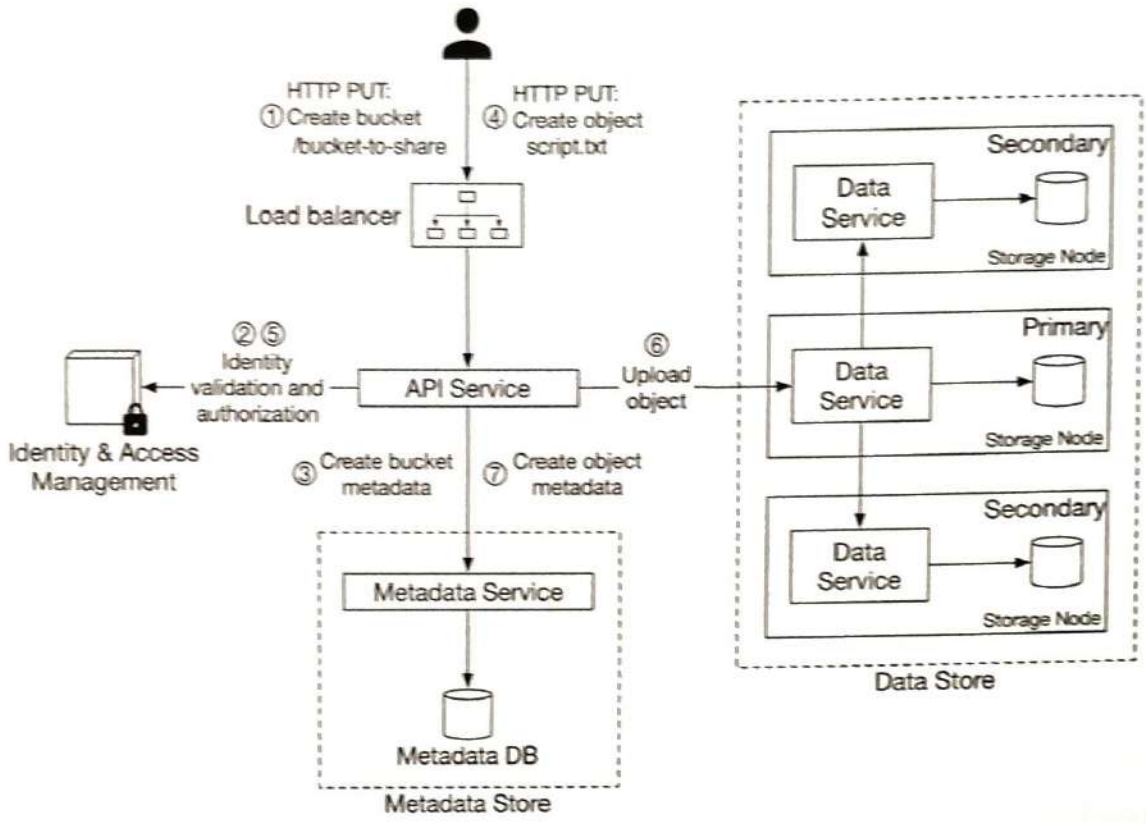


Figure 9.5: Uploading an object

An object has to reside in a bucket. In this example, we first create a bucket named `bucket-to-share` and then upload a file named `script.txt` to the bucket. Figure 9.5 explains how this flow works in 7 steps.

1. The client sends an `HTTP PUT` request to create a bucket named `bucket-to-share`. The

request is forwarded to the API service.

2. The API service calls the IAM to ensure the user is authorized and has `WRITE` permission.
3. The API service calls the metadata store to create an entry with the bucket info in the metadata database. Once the entry is created, a success message is returned to the client.
4. After the bucket is created, the client sends an `HTTP PUT` request to create an object named `script.txt`.
5. The API service verifies the user's identity and ensures the user has `WRITE` permission on the bucket.
6. Once validation succeeds, the API service sends the object data in the `HTTP PUT` payload to the data store. The data store persists the payload as an object and returns the UUID of the object.
7. The API service calls the metadata store to create a new entry in the metadata database. It contains important metadata such as the `object_id` (UUID), `bucket_id` (which bucket the object belongs to), `object_name`, etc. A sample entry is shown in Table 9.2.

object_name	object_id	bucket_id
script.txt	239D5866-0052-00F6-014E-C914E61ED42B	82AA1B2E-F599-4590-B5E4-1F51AAE5F7E4

Table 9.2: Sample entry

The API to upload an object could look like this:

```
PUT /bucket-to-share/script.txt HTTP/1.1
Host: foo.s3example.org
Date: Sun, 12 Sept 2021 17:51:00 GMT
Authorization: authorization string
Content-Type: text/plain
Content-Length: 4567
x-amz-meta-author: Alex

[4567 bytes of object data]
```

Listing 9.2: Uploading an object

Downloading an object

A bucket has no directory hierarchy. However, we can create a logical hierarchy by concatenating the bucket name and the object name to simulate a folder structure. For example, we name the object `bucket-to-share/script.txt` instead of `script.txt`. To get an object, we specify the object name in the `GET` request. The API to download an object looks like this:

```
GET /bucket-to-share/script.txt HTTP/1.1
Host: foo.s3example.org
Date: Sun, 12 Sept 2021 18:30:01 GMT
Authorization: authorization string
```

Listing 9.3: Downloading an object

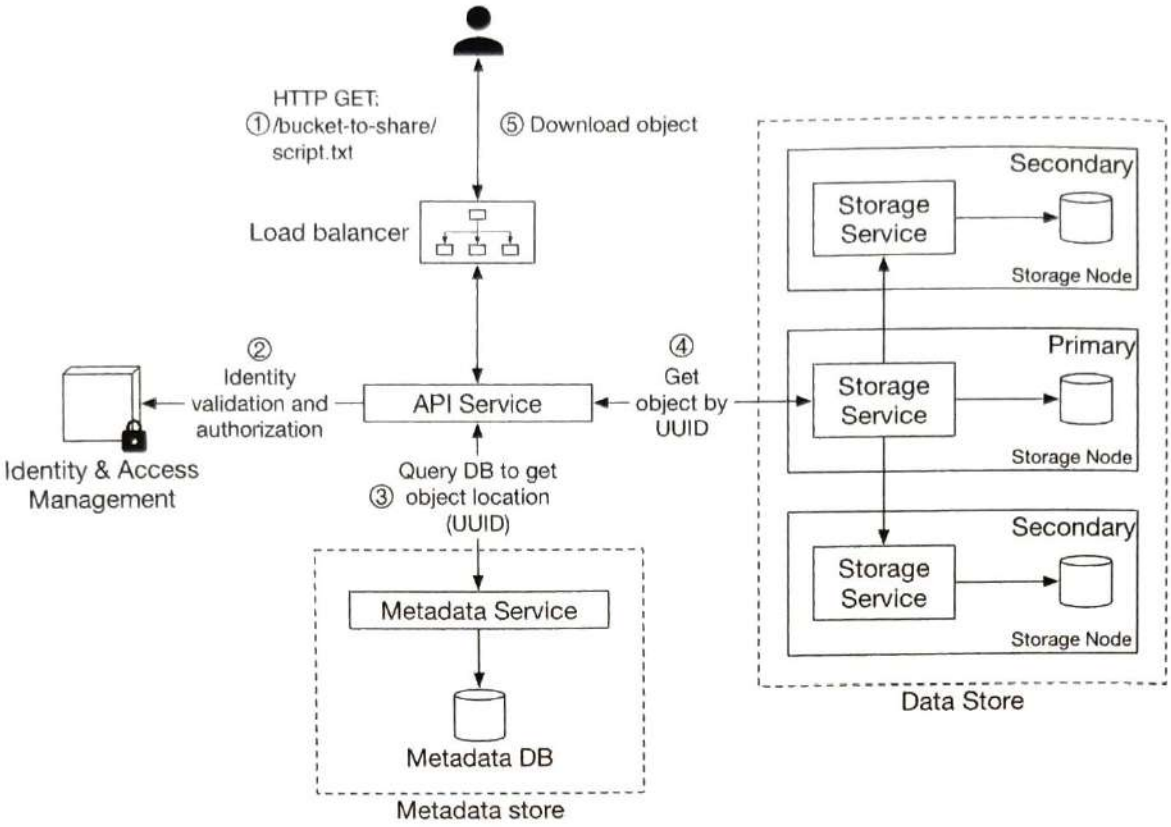


Figure 9.6: Downloading an object

As mentioned earlier, the data store does not store the name of the object and it only supports object operations via `object_id` (UUID). In order to download the object, we first map the object name to the UUID. The workflow of downloading an object is shown below:

1. The client sends an `HTTP GET` request to the load balancer: `GET /bucket-to-share/script.txt`
2. The API service queries the IAM to verify that the user has `READ` access to the bucket.
3. Once validated, the API service fetches the corresponding object's UUID from the metadata store.
4. Next, the API service fetches the object data from the data store by its UUID.
5. The API service returns the object data to the client in HTTP GET response.

Step 3 - Design Deep Dive

In this section, we dive deep into a few areas:

- Data store
- Metadata data model
- Listing objects in a bucket
- Object versioning
- Optimizing uploads of large files
- Garbage collection

Data store

Let's take a closer look at the design of the data store. As discussed previously, the API service handles external requests from users and calls different internal services to fulfill those requests. To persist or retrieve an object, the API service calls the data store. Figure 9.7 shows the interactions between the API service and the data store for uploading and downloading an object.

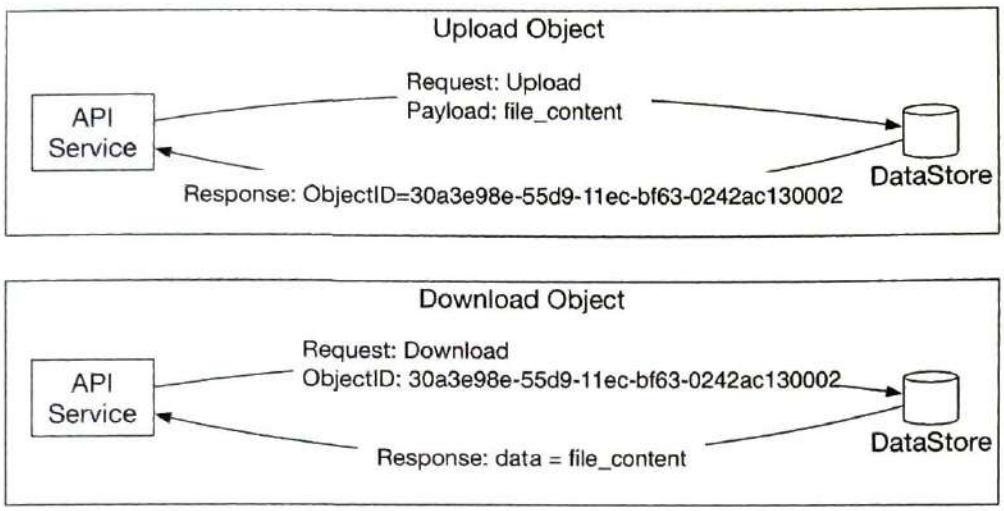


Figure 9.7: Upload and download an object

High-level design for the data store

The data store has three main components as shown in Figure 9.8.

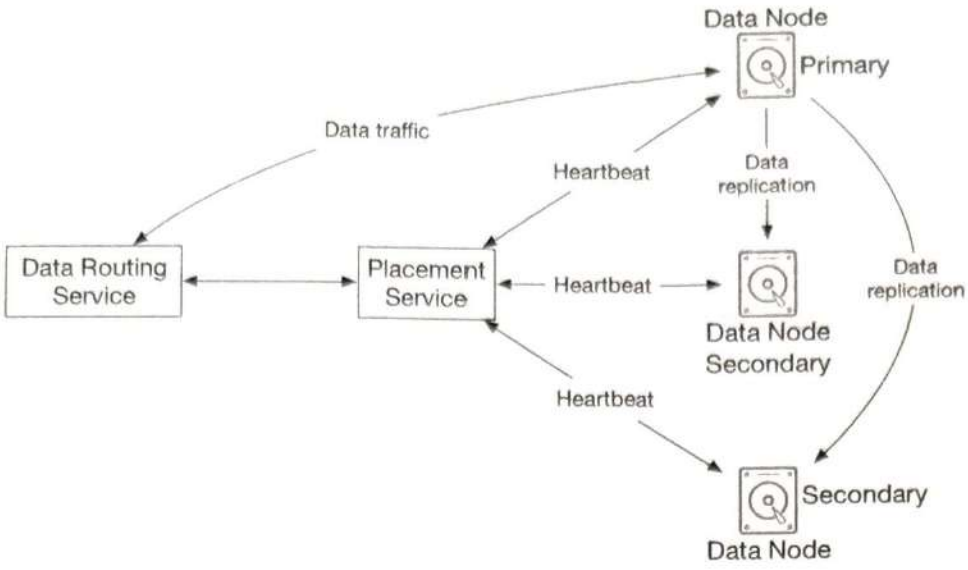


Figure 9.8: Data store components

Data routing service

The data routing service provides RESTful or gRPC [12] APIs to access the data node cluster. It is a stateless service that can scale by adding more servers. This service has the following responsibilities:

- Query the placement service to get the best data node to store data.
- Read data from data nodes and return it to the API service.
- Write data to data nodes.

Placement service

The placement service determines which data nodes (primary and replicas) should be chosen to store an object. It maintains a virtual cluster map, which provides the physical topology of the cluster. The virtual cluster map contains location information for each data node which the placement service uses to make sure the replicas are physically separated. This separation is key to high durability. See the "Durability" section on page 270 for details. An example of the virtual cluster map is shown in Figure 9.9.

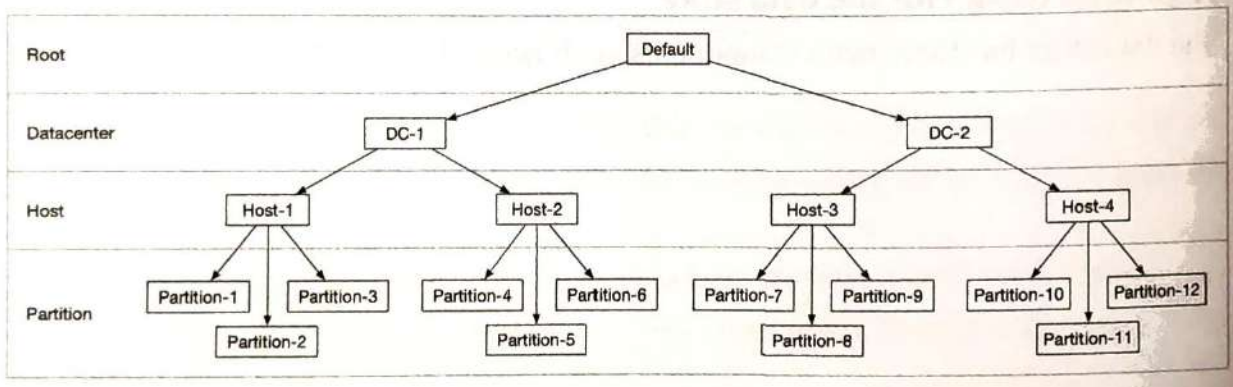


Figure 9.9: Virtual cluster map

The placement service continuously monitors all data nodes through heartbeats. If a data node doesn't send a heartbeat within a configurable 15-second grace period, the placement service marks the node as "down" in the virtual cluster map.

This is a critical service, so we suggest building a cluster of 5 or 7 placement service nodes with Paxos [13] or Raft [14] consensus protocol. The consensus protocol ensures that as long as more than half of the nodes are healthy, the service as a whole continues to work. For example, if the placement service cluster has 7 nodes, it can tolerate a 3 node failure. To learn more about consensus protocols, refer to the reference materials [13] [14].

Data node

The data node stores the actual object data. It ensures reliability and durability by replicating data to multiple data nodes, also called a replication group.

Each data node has a data service daemon running on it. The data service daemon continuously sends heartbeats to the placement service. The heartbeat message includes the following essential information:

- How many disk drives (HDD or SSD) does the data node manage?
- How much data is stored on each drive?

When the placement service receives the heartbeat for the first time, it assigns an ID for this data node, adds it to the virtual cluster map, and returns the following information:

- a unique ID of the data node
- the virtual cluster map
- where to replicate data

Data persistence flow

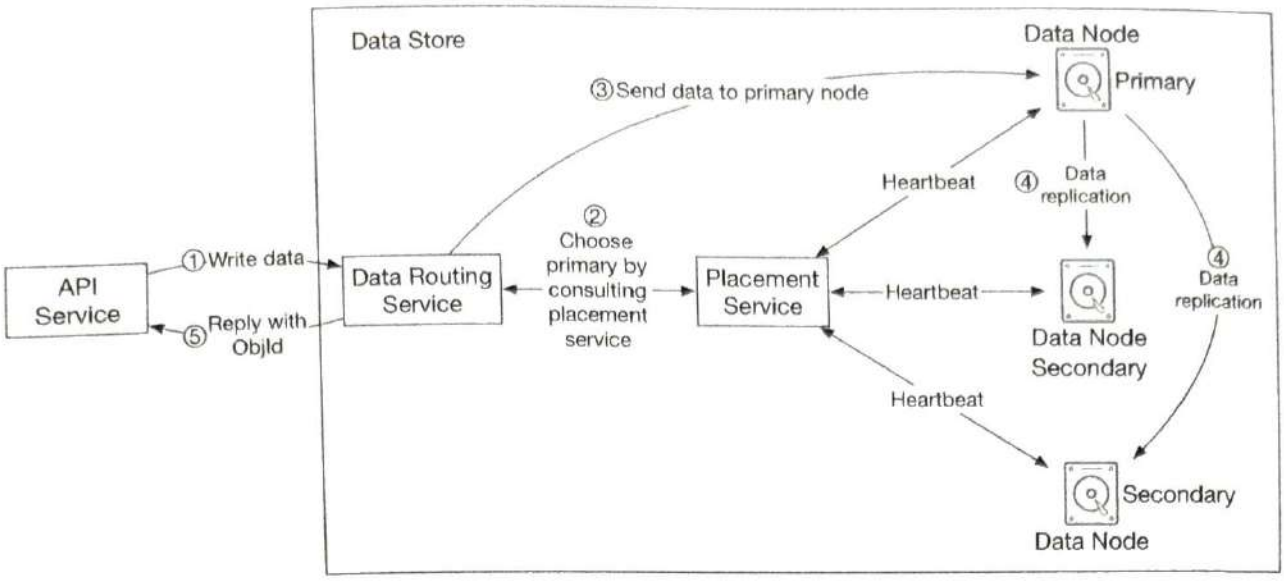


Figure 9.10: Data persistence flow

Now let's take a look at how data is persisted in the data node.

1. The API service forwards the object data to the data store.
2. The data routing service generates a UUID for this object and queries the placement service for the data node to store this object. The placement service checks the virtual cluster map and returns the primary data node.
3. The data routing service sends data directly to the primary data node, together with its UUID.
4. The primary data node saves the data locally and replicates it to two secondary data nodes. The primary node responds to the data routing service when data is successfully replicated to all secondary nodes.
5. The UUID of the object (ObjId) is returned to the API service.

In step 2, given a UUID for the object as an input, the placement service returns the replication group for the object. How does the placement service do this? Keep in mind that this lookup needs to be deterministic, and it must survive the addition or removal of replication groups. Consistent hashing is a common implementation of such a lookup function. Refer to [15] for more information.

In step 4, the primary data node replicates data to all secondary nodes before it returns a response. This makes data strongly consistent among all data nodes. This consistency comes with latency costs because we have to wait until the slowest replica finishes. Figure 9.11 shows the trade-offs between consistency and latency.

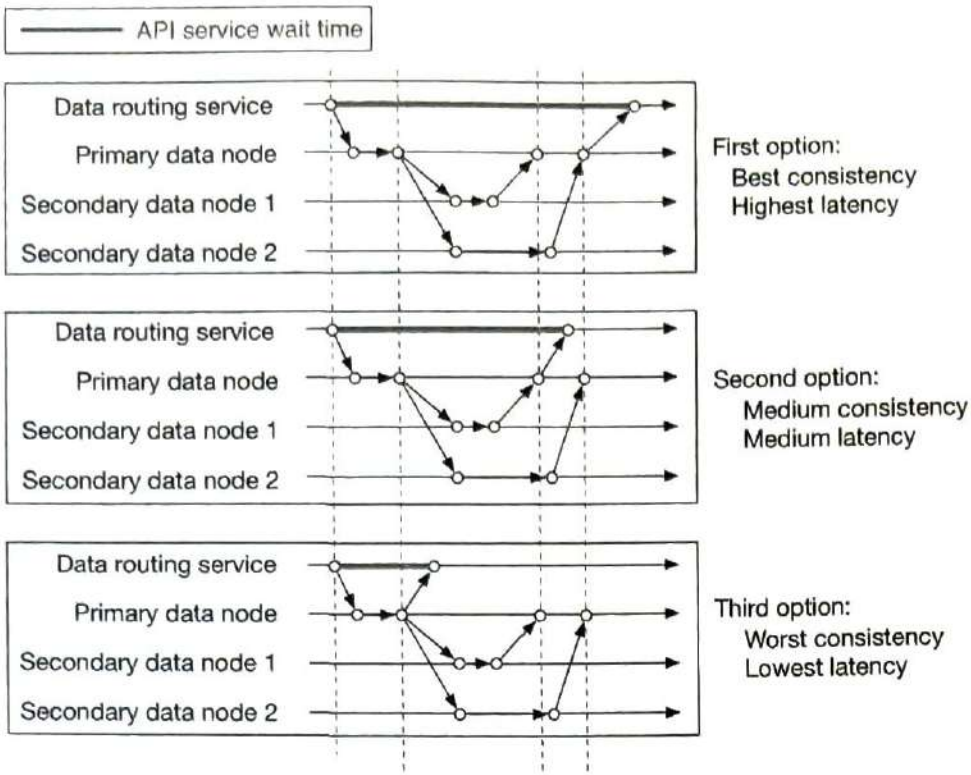


Figure 9.11: Trade-off between consistency and latency

1. Data is considered as successfully saved after all three nodes store the data. This approach has the best consistency but the highest latency.
2. Data is considered as successfully saved after the primary and one of the secondaries store the data. This approach has a medium consistency and medium latency.
3. Data is considered as successfully saved after the primary persists the data. This approach has the worst consistency but the lowest latency.

Both 2 and 3 are forms of eventual consistency.

How data is organized

Now let's take a look at how each data node manages the data. A simple solution is to store each object in a stand-alone file. This works, but the performance suffers when there are many small files. Two issues arise when having too many small files on a file system. First, it wastes many data blocks. A file system stores files in discrete disk blocks. Disk blocks have the same size, and the size is fixed when the volume is initialized. The typical block size is around 4KB. For a file smaller than 4KB, it would still consume the entire disk block. If the file system holds a lot of small files, it wastes a lot of disk blocks, with each one only lightly filled with a small file.

Second, it could exceed the system's inode capacity. The file system stores the location and other information about a file in a special type of block called inode. For most file systems, the number of inodes is fixed when the disk is initialized. With millions of small files, it runs the risk of consuming all inodes. Also, the operating system does not handle a large number of inodes very well, even with aggressive caching of file system

metadata. For these reasons, storing small objects as individual files does not work well in practice.

To address these issues, we can merge many small objects into a larger file. It works conceptually like a write-ahead log (WAL). When we save an object, it is appended to an existing read-write file. When the read-write file reaches its capacity threshold (usually set to a few GBs), the read-write file is marked as read-only and a new read-write file is created to receive new objects. Once a file is marked as read-only, it can only serve read requests. Figure 9.12 explains how this process works.

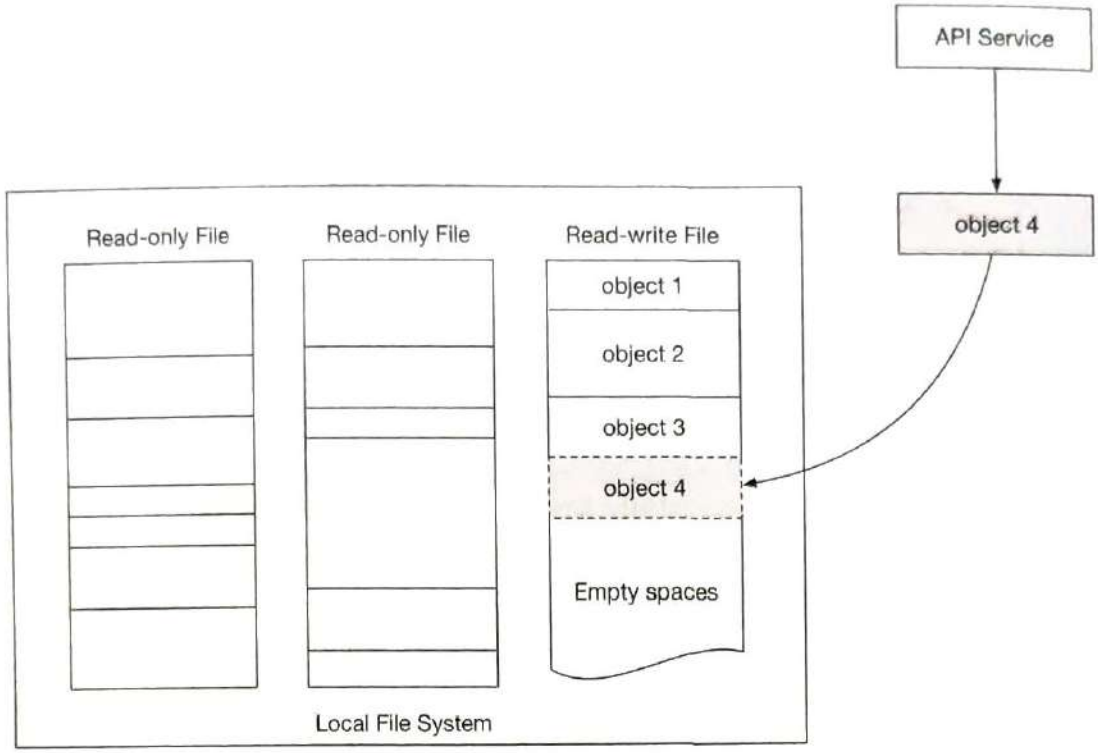


Figure 9.12: Store multiple small objects in one big file

Note that write access to the read-write file must be serialized. As shown in Figure 9.12, objects are stored in order, one after the other, in the read-write file. To maintain this on-disk layout, multiple cores processing incoming write requests in parallel must take their turns to write to the read-write file. For a modern server with a large number of cores processing many incoming requests in parallel, this seriously restricts write throughput. To fix this, we could provide dedicated read-write files, one for each core processing incoming requests.

Object lookup

With each data file holding many small objects, how does the data node locate an object by UUID? The data node needs the following information:

- The data file that contains the object
- The starting offset of the object in the data file
- The size of the object

The database schema to support this lookup is shown in Table 9.3.

object_mapping
object_id
file_name
start_offset
object_size

Table 9.3: `Object_mapping` table

Field	Description
`object_id`	UUID of the object
`file_name`	The name of the file that contains the object
`start_offset`	Beginning address of the object in the file
`object_size`	The number of bytes in the object

Table 9.4: `Object_mapping` fields

We considered two options for storing this mapping: a file-based key-value store such as RocksDB [16] or a relational database. RocksDB is based on SSTable [17], and it is fast for writes but slower for reads. A relational database usually uses a B+ tree [18] based storage engine, and it is fast for reads but slower for writes. As mentioned earlier, the data access pattern is write once and read multiple times. Since a relational database provides better read performance, it is a better choice than RocksDB.

How should we deploy this relational database? At our scale, the data volume for the mapping table is massive. Deploying a single large cluster to support all data nodes could work, but is difficult to manage. Note that this mapping data is isolated within each data node. There is no need to share this across data nodes. To take advantage of this property, we could simply deploy a simple relational database on each data node. SQLite [19] is a good choice here. It is a file-based relational database with a solid reputation.

Updated data persistence flow

Since we have made quite a few changes to the data node, let's revisit how to save a new object in the data node (Figure 9.13).

1. The API service sends a request to save a new object named `object 4`.
2. The data node service appends the object named `object 4` at the end of the read-write file named `/data/c`.
3. A new record of `object 4` is inserted into the `object_mapping` table.
4. The data node service returns the UUID to the API service.

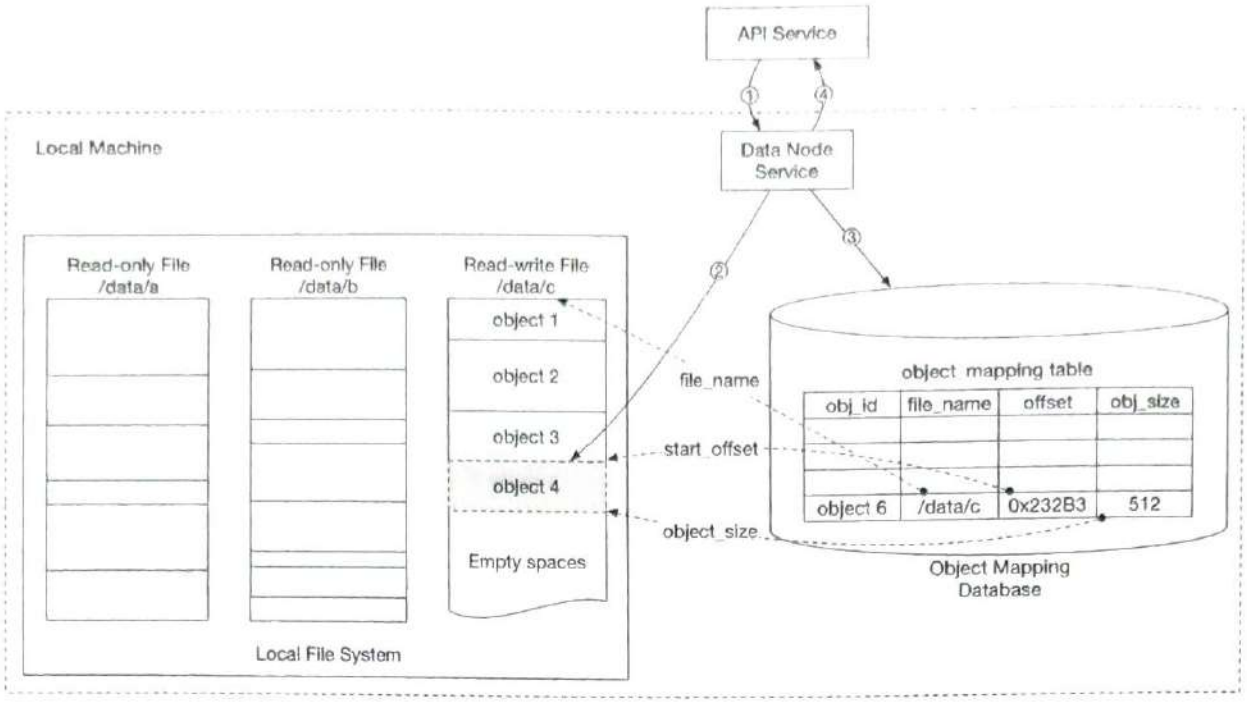


Figure 9.13: Updated data persistence flow

Durability

Data reliability is extremely important for data storage systems. How can we create a storage system that offers six nines of durability? Each failure case has to be carefully considered and data needs to be properly replicated.

Hardware failure and failure domain

Hard drive failures are inevitable no matter which media we use. Some storage media may have better durability than others, but we cannot rely on a single hard drive to achieve our durability objective. A proven way to increase durability is to replicate data to multiple hard drives, so a single disk failure does not impact the data availability, as a whole. In our design, we replicate data three times.

Let's assume the spinning hard drive has an annual failure rate of 0.81% [20]. This number highly depends on the model and make. Making 3 copies of data gives us $1 - 0.0081^3 =\sim 0.999999$ reliability. This is a very rough estimate. For more sophisticated calculations, please read [20].

For a complete durability evaluation, we also need to consider the impacts of different failure domains. A failure domain is a physical or logical section of the environment that is negatively affected when a critical service experiences problems. In a modern data center, a server is usually put into a rack [21], and the racks are grouped into rows/floors/-rooms. Since each rack shares network switches and power, all the servers in a rack are in a rack-level failure domain. A modern server shares components like the motherboard, processors, power supply, HDD drives, etc. The components in a server are in a node-level failure domain.

Here is a good example of a large-scale failure domain isolation. Typically, data cen-

ters divide infrastructure that shares nothing into different Availability Zones (AZs). We replicate our data to different AZs to minimize the failure impact (Figure 9.14). Note that the choice of failure domain level doesn't directly increase the durability of data, but it will result in better reliability in extreme cases, such as large-scale power outages, cooling system failures, natural disasters, etc.

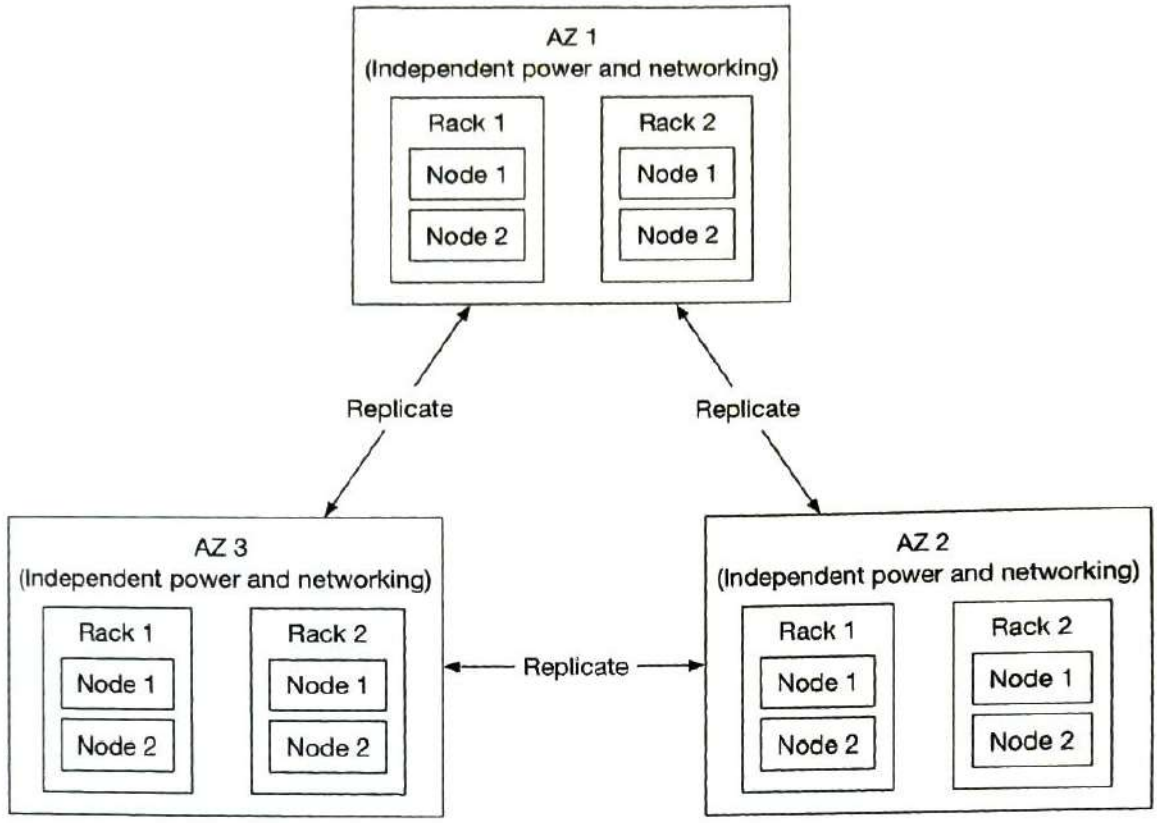


Figure 9.14: Multi-Datacenter replication

Erasure coding

Making three full copies of data gives us roughly 6 nines of data durability. Are there other options to further increase durability? Yes, erasure coding is one option. Erasure coding [22] deals with data durability differently. It chunks data into smaller pieces (placed on different servers) and creates parities for redundancy. In the event of failures, we can use chunk data and parities to reconstruct the data. Let's take a look at a concrete example ($4 + 2$ erasure coding) as shown in Figure 9.15.

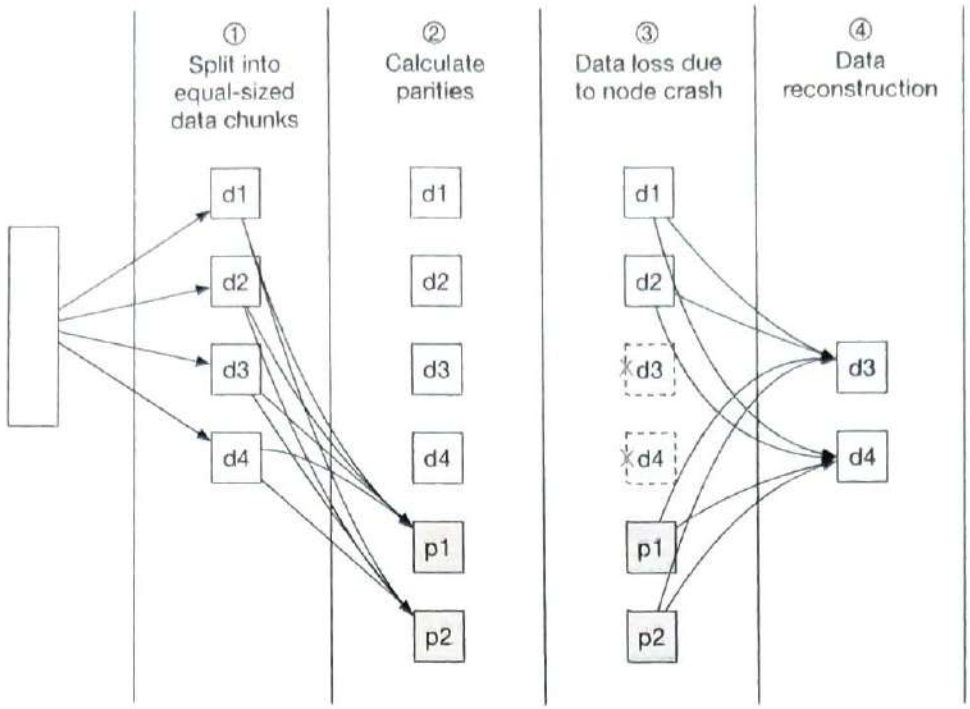


Figure 9.15: Erasure coding

1. Data is broken up into four even-sized data chunks d1, d2, d3, and d4.
2. The mathematical formula [23] is used to calculate the parities p1 and p2. To give a much simplified example, $p1 = d1 + 2 \times d2 - d3 + 4 \times d4$ and $p2 = -d1 + 5 \times d2 + d3 - 3 \times d4$ [24].
3. Data d3 and d4 are lost due to node crashes.
4. The mathematical formula is used to reconstruct lost data d3 and d4, using the known values of d1, d2, p1, and p2.

Let's take a look at another example as shown in Figure 9.16 to better understand how erasure coding works with failure domains. An (8+4) erasure coding setup breaks up the original data evenly into 8 chunks and calculates 4 parities. All 12 pieces of data have the same size. All 12 chunks of data are distributed across 12 different failure domains. The mathematics behind erasure coding ensures that the original data can be reconstructed when at most 4 nodes are down.

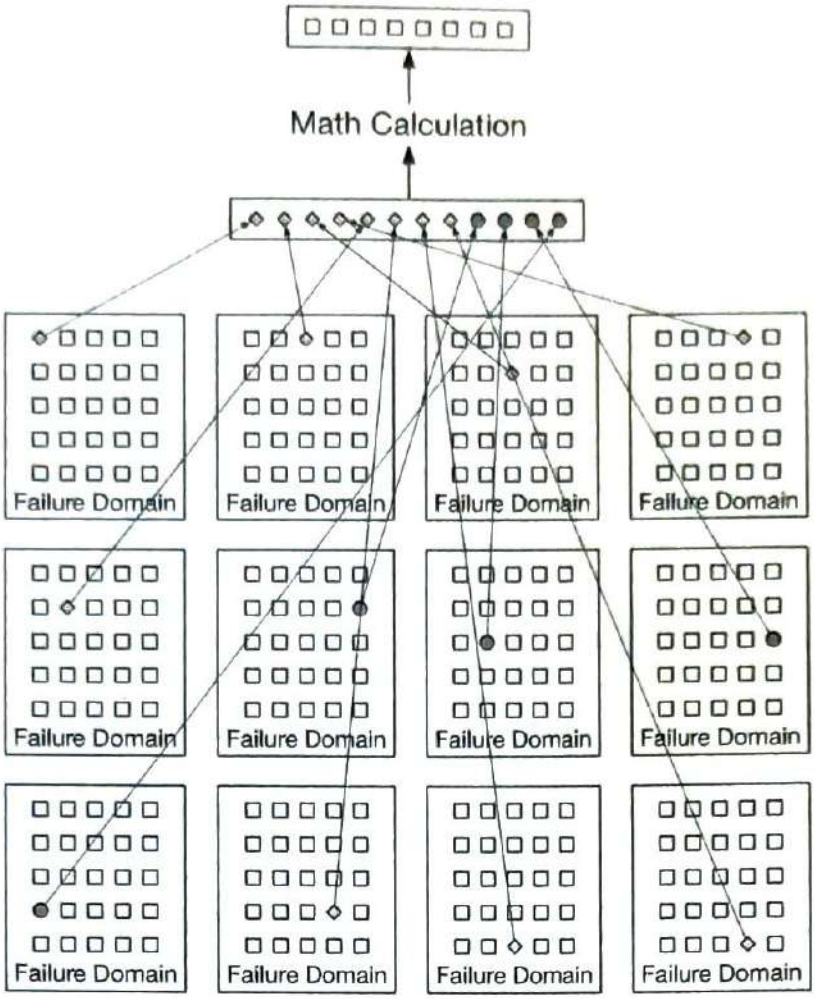


Figure 9.16: (8 + 4) erasure coding

Compared to replication where the data router only needs to read data for an object from one healthy node, in erasure coding the data router has to read data from at least 8 healthy nodes. This is an architectural design tradeoff. We use a more complex solution with a slower access speed, in exchange for higher durability and lower storage cost. For object storage where the main cost is storage, this tradeoff might be worth it.

How much extra space does erasure coding need? For every two chunks of data, we need one parity block, so the storage overhead is 50% (Figure 9.17). While in 3-copy replication, the storage overhead is 200% (Figure 9.17).

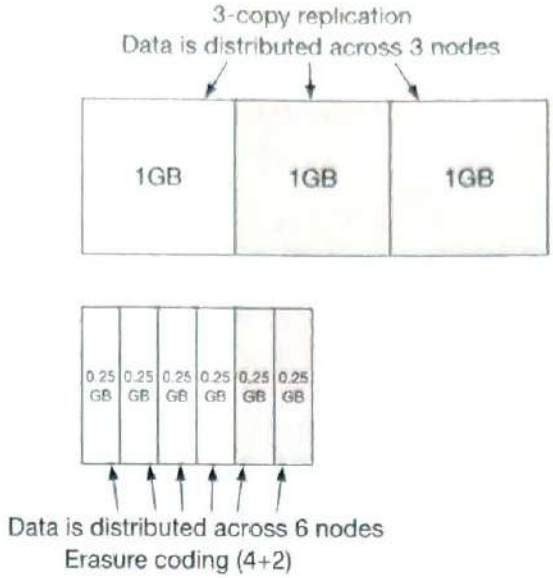


Figure 9.17: Extra space for replication and erasure coding

Does erasure coding increase data durability? Let's assume a node has a 0.81% annual failure rate. According to the calculation done by Backblaze [20], erasure coding can achieve 11 nines durability. The calculation requires complicated math. If you're interested, refer to [20] for details.

Table 9.5 compares the pros and cons of replication and erasure coding.

	Replication	**Erasure coding**
Durability	6 nines of durability (data copied 3 times)	11 nines of durability (8 + 4 erasure coding). **Erasure coding wins.**
Storage efficiency	200% storage overhead.	50% storage overhead. **Erasure coding wins.**
Compute resource	No computation. **Replication wins.**	Higher usage of computation resources to calculate parities.
Write performance	Replicating data to multiple nodes. No calculation is needed. **Replication wins.**	Increased write latency because we need to calculate parities before data is written to disk.
Read performance	In normal operation, reads are served from the same replica. Reads under a failure mode are not impacted because reads can be served from a non-fault replica. **Replication wins.**	In normal operation, every read has to come from multiple nodes in the cluster. Reads under a failure mode are slower because the missing data must be reconstructed first.

Table 9.5: Replication vs erasure coding

In summary, replication is widely adopted in latency-sensitive applications while erasure coding is often used to minimize storage cost. Erasure coding is attractive for its cost

efficiency and durability, but it greatly complicates the data node design. Therefore, for this design, we mainly focus on replication.

Correctness verification

Erasure coding increases data durability at comparable storage costs. Now we can move on to solve another hard challenge: data corruption.

If a disk fails completely and the failure can be detected, it can be treated as a data node failure. In this case, we can reconstruct data using erasure coding. However, in-memory data corruption is a regular occurrence in large-scale systems.

This problem can be addressed by verifying checksums [25] between process boundaries. A checksum is a small-sized block of data that is used to detect data errors. Figure 9.18 illustrates how the checksum is generated.

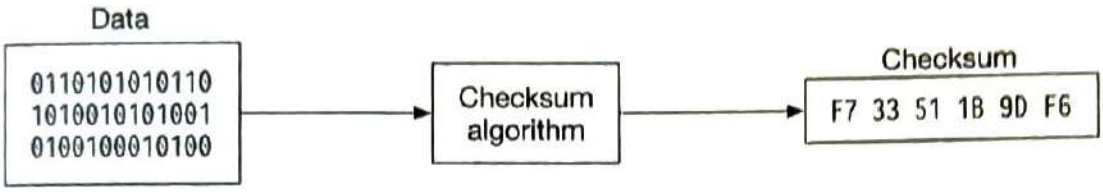


Figure 9.18: Generate checksum

If we know the checksum of the original data, we can compute the checksum of the data after transmission:

- If they are different, data is corrupted.
- If they are the same, there is a very high probability the data is not corrupted. The probability is not 100%, but in practice, we could assume they are the same.

Figure 9.19: Compare checksums

There are many checksum algorithms, such as MD5 [26], SHA1 [27], HMAC [28], etc. A good checksum algorithm usually outputs a significantly different value even for a small change made to the input. For this chapter, we choose a simple checksum algorithm such as MD5.

In our design, we append the checksum at the end of each object. Before a file is marked as read-only, we add a checksum of the entire file at the end. Figure 9.20 shows the layout.

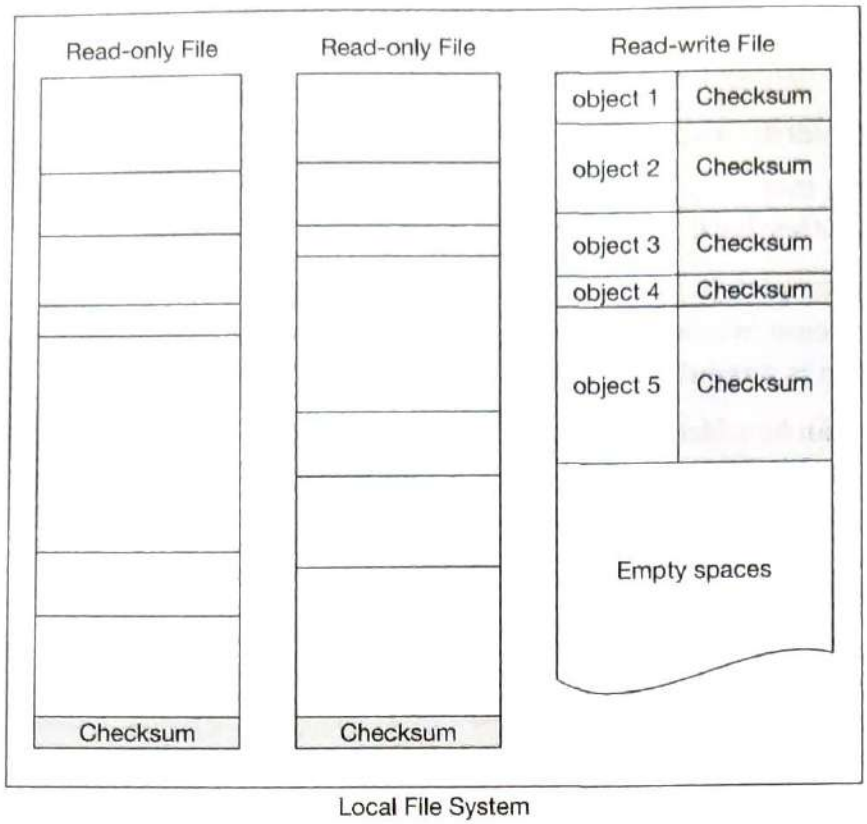


Figure 9.20: Add checksum to data node

With $(8 + 4)$ erasure coding and checksum verification, this is what happens when we read data:

1. Fetch the object data and the checksum.
2. Compute the checksum against the data received.
   (a) If the two checksums match, the data is error-free.
   (b) If the checksums are different, the data is corrupted. We will try to recover by reading the data from other failure domains.
3. Repeat steps 1 and 2 until all 8 pieces of data are returned. We then reconstruct the data and send it back to the client.

Metadata data model

In this section, we first discuss the database schema and then dive into scaling the database.

Schema

The database schema needs to support the following 3 queries:

Query 1: Find the object ID by object name.

Query 2: Insert and delete an object based on the object name.

Query 3: List objects in a bucket sharing the same prefix.

Figure 9.21 shows the schema design. We need two database tables: `bucket` and `object`.

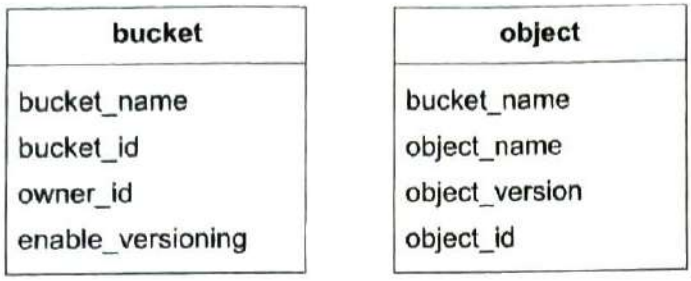


Figure 9.21: Database tables

Scale the `bucket` table

Since there is usually a limit on the number of buckets a user can create, the size of the `bucket` table is small. Let's assume we have 1 million customers, each customer owns 10 buckets and each record takes 1KB. That means we need 10GB (1 million $\times 10 \times$ 1KB) of storage space. The whole table can easily fit in a modern database server. However, a single database server might not have enough CPU or network bandwidth to handle all read requests. If so, we can spread the read load among multiple database replicas.

Scale the `object` table

The `object` table holds the object metadata. The dataset at our design scale will likely not fit in a single database instance. We can scale the object table by sharding.

One option is to shard by the `bucket_id` so all the objects under the same bucket are stored in one shard. This doesn't work because it causes hotspot shards as a bucket might contain billions of objects.

Another option is to shard by `object_id`. The benefit of this sharding scheme is that it evenly distributes the load. But we will not be able to execute query 1 and query 2 efficiently because those two queries are based on the URI.

We choose to shard by a combination of `bucket_name` and `object_name`. This is because most of the metadata operations are based on the object URI, for example, finding the object ID by URI or uploading an object via URI. To evenly distribute the data, we can use the hash of the `<bucket_name, object_name>` as the sharding key.

With this sharding scheme, it is straightforward to support the first two queries, but the last query is less obvious. Let's take a look.

Listing objects in a bucket

The object store arranges files in a flat structure instead of a hierarchy, like in a file system. An object can be accessed using a path in this format, `s3://bucket-name/object-name`. For example, `s3://mybucket/abc/d/e/f/file.txt` contains:

- Bucket name: `mybucket`
- Object name: `abc/d/e/f/file.txt`

To help users organize their objects in a bucket, S3 introduces a concept called 'prefixes'. A prefix is a string at the beginning of the object name. S3 uses prefixes to organize the data in a way similar to directories. However, prefixes are not directories. Listing a bucket by prefix limits the results to only those object names that begin with the prefix.

In the example above with the path `s3://mybucket/abc/d/e/f/file.txt`, the prefix is `abc/d/e/f/`.

The AWS S3 listing command has 3 typical uses:

1. List all buckets owned by a user. The command looks like this:

   ```
   aws s3 list-buckets
   ```

2. List all objects in a bucket that are at the same level as the specified prefix. The command looks like this:

   ```
   aws s3 ls s3://mybucket/abc/
   ```

   In this mode, objects with more slashes in the name after the prefix are rolled up into a common prefix. For example, with these objects in the bucket:

   ```
   CA/cities/losangeles.txt
   CA/cities/sanfranciso.txt
   NY/cities/ny.txt
   federal.txt
   ```

   Listing the bucket with the "/" prefix would return these results, with everything under `CA/` and `NY/` rolled up into them:

   ```
   CA/
   NY/
   federal.txt
   ```

3. Recursively list all objects in a bucket that shares the same prefix. The command looks like this:

   ```
   aws s3 ls s3://mybucket/abc/ --recursive
   ```

   Using the same example as above, listing the bucket with the `CA/` prefix would return these results:

   ```
   CA/cities/losangeles.txt
   CA/cities/sanfranciso.txt
   ```

Single database

Let's first explore how we would support the listing command with a single database. To list all buckets owned by a user, we run the following query:

```
SELECT * FROM bucket WHERE owner_id={id}
```

To list all objects in a bucket that share the same prefix, we run a query like this.

```
SELECT * FROM object
WHERE bucket_id = "123" AND object_name LIKE `abc/%`
```

In this example, we find all objects with bucket_id equals to 123 that share the prefix abc/. Any objects with more slashes in their names after the specified prefix are rolled up in the application code as stated earlier in use case 2.

The same query would support the recursive listing mode, as stated in use case 3 previously. The application code would list every object sharing the same prefix, without performing any rollups.

Distributed databases

When the metadata table is sharded, it's difficult to implement the listing function because we don't know which shards contain the data. The most obvious solution is to run a search query on all shards and then aggregate the results. To achieve this, we can do the following:

1. The metadata service queries every shard by running the following query:

```
SELECT * FROM object
WHERE bucket_id = "123" AND object_name LIKE `a/b/%`
```

2. The metadata service aggregates all objects returned from each shard and returns the result to the caller.

This solution works, but implementing pagination for this is a bit complicated. Before we explain why, let's review how pagination works for a simple case with a single database. To return pages of listing with 10 objects for each page, the SELECT query would start with this:

```
SELECT * FROM object
WHERE bucket_id = "123" AND object_name LIKE `a/b/%`
ORDER BY object_name OFFSET 0 LIMIT 10
```

The OFFSET and LIMIT would restrict the results to the first 10 objects. In the next call, the user sends the request with a hint to the server, so it knows to construct the query for the second page with an OFFSET of 10. This hint is usually done with a cursor that the server returns with each page to the client. The offset information is encoded in the cursor. The client would include the cursor in the request for the next page. The server decodes the cursor and uses the offset information embedded in it to construct the query for the next page. To continue with the example above, the query for the second page looks like this:

```
SELECT * FROM metadata
WHERE bucket_id = "123" AND object_name LIKE `a/b/%`
ORDER BY object_name OFFSET 10 LIMIT 10
```

This client-server request loop continues until the server returns a special cursor that marks the end of the entire listing.

Now, let's explore why it's complicated to support pagination for sharded databases. Since the objects are distributed across shards, the shards would likely return a varying number of results. Some shards would contain a full page of 10 objects, while others

would be partial or empty. The application code would receive results from every shard, aggregate and sort them, and return only a page of 10 in our example. The objects that don't get included in the current round must be considered again for the next round. This means that each shard would likely have a different offset. The server must track the offsets for all the shards and associate those offsets with the cursor. If there are hundreds of shards, there will be hundreds of offsets to track.

We have a solution that can solve the problem, but there are some tradeoffs. Since object storage is tuned for vast scale and high durability, object listing performance is rarely a priority. In fact, all commercial object storage supports object listing with sub-optimal performance. To take advantage of this, we could denormalize the listing data into a separate table sharded by bucket ID. This table is only used for listing objects. With this setup, even buckets with billions of objects would offer acceptable performance. This isolates the listing query to a single database which greatly simplifies the implementation.

Object versioning

Versioning is a feature that keeps multiple versions of an object in a bucket. With versioning, we can restore objects that are accidentally deleted or overwritten. For example, we may modify a document and save it under the same name, inside the same bucket. Without versioning, the old version of the document metadata is replaced by the new version in the metadata store. The old version of the document is marked as deleted, so its storage space will be reclaimed by the garbage collector. With versioning, the object storage keeps all previous versions of the document in the metadata store, and the old versions of the document are never marked as deleted in the object store.

Figure 9.22 explains how to upload a versioned object. For this to work, we first need to enable versioning on the bucket.

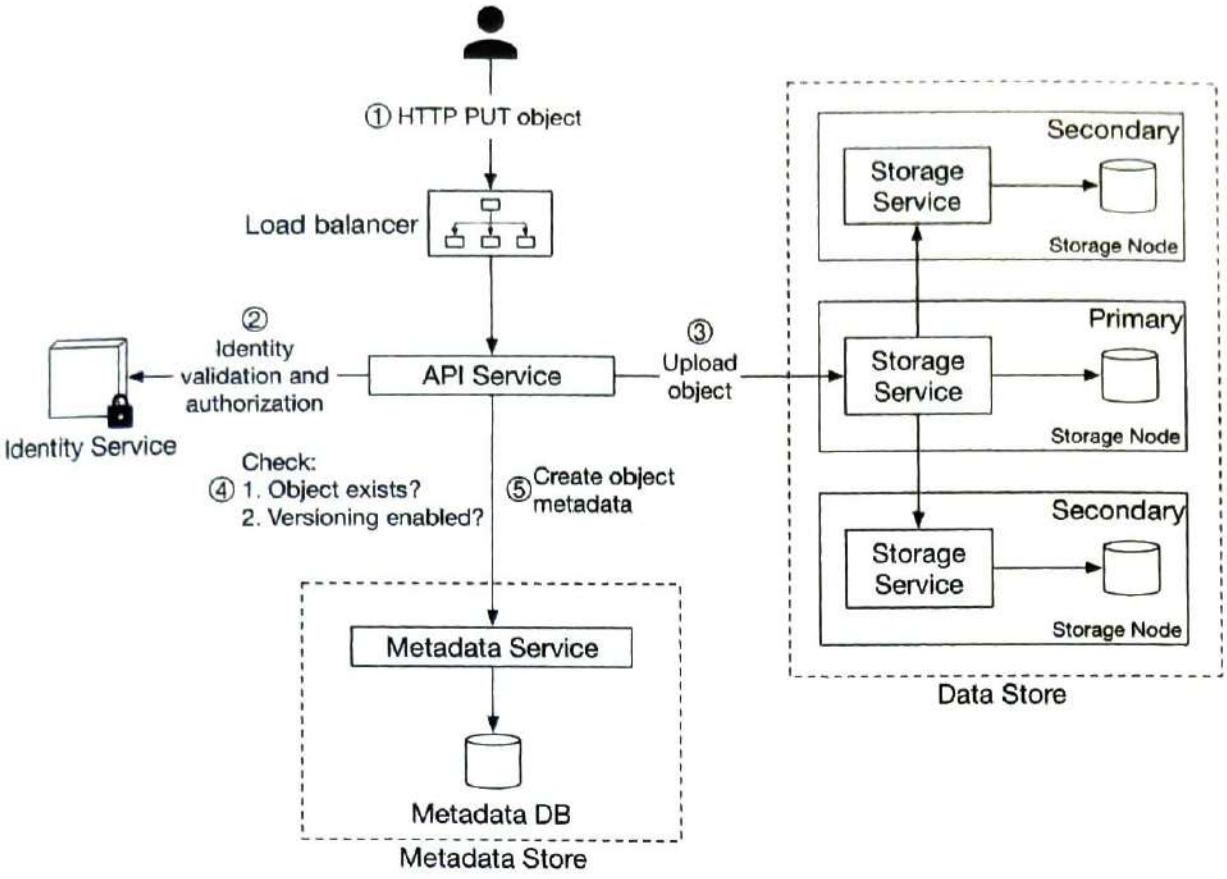


Figure 9.22: Object versioning

1. The client sends an `HTTP PUT` request to upload an object named `script.txt`.
2. The API service verifies the user's identity and ensures that the user has `WRITE` permission on the bucket.
3. Once verified, the API service uploads the data to the data store. The data store persists the data as a new object and returns a new UUID to the API service.
4. The API service calls the metadata store to store the metadata information of this object.
5. To support versioning, the object table for the metadata store has a column called `object_version` that is only used if versioning is enabled. Instead of overwriting the existing record, a new record is inserted with the same `bucket_id` and `object_name` as the old record, but with a new `object_id` and `object_version`. The `object_id` is the UUID for the new object returned in step 3. The `object_version` is a `TIMEUUID` [29] generated when the new row is inserted. No matter which database we choose for the metadata store, it should be efficient to look up the current version of an object. The current version has the largest `TIMEUUID` of all the entries with the same `object_name`. See Figure 9.23 for an illustration of how we store versioned metadata.

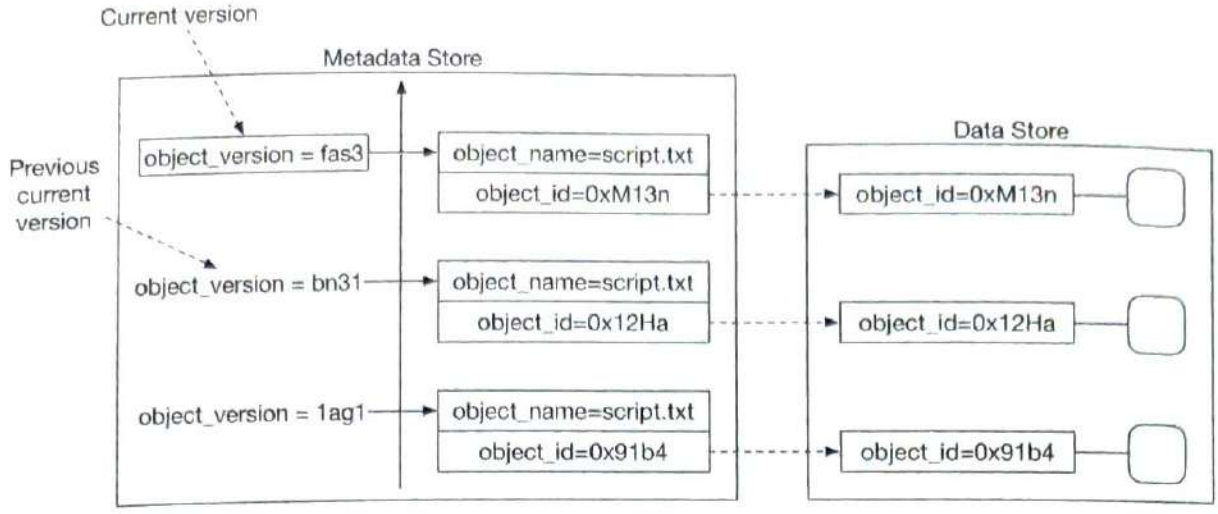


Figure 9.23: Versioned metadata

In addition to uploading a versioned object, it can also be deleted. Let's take a look.

When we delete an object, all versions remain in the bucket and we insert a delete marker, as shown in Figure 9.24.

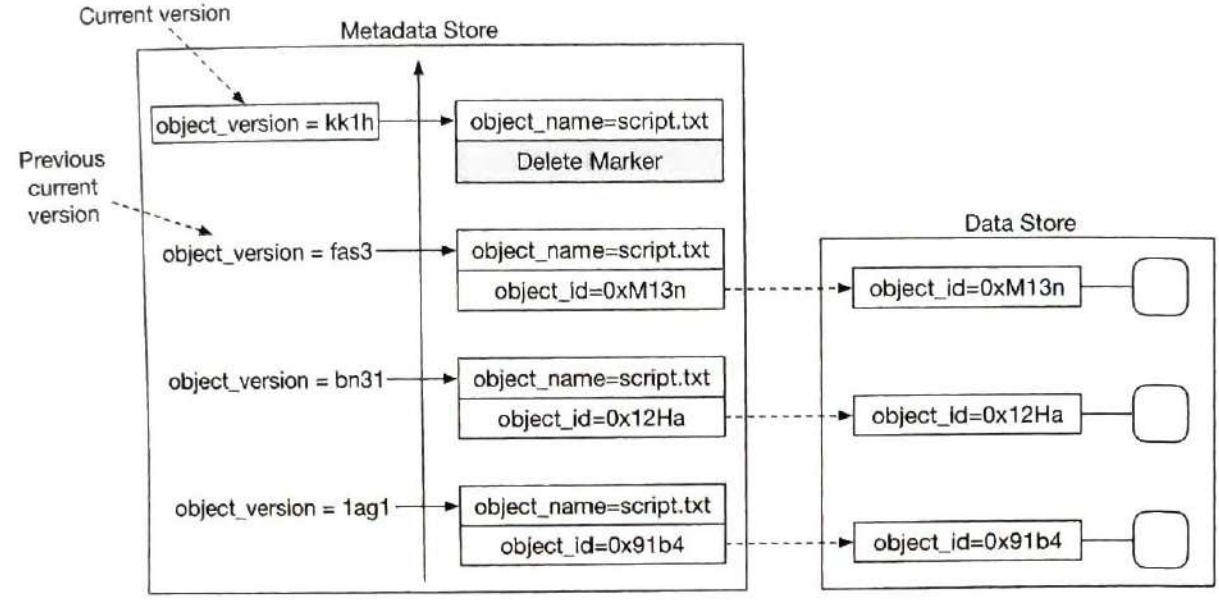


Figure 9.24: Delete object by inserting a delete marker

A delete marker is a new version of the object, and it becomes the current version of the object once inserted. Performing a `GET` request when the current version of the object is a delete marker returns a `404 Object Not Found` error.

Optimizing uploads of large files

In the back-of-the-envelope estimation, we estimated that 20% of the objects are large. Some might be larger than a few GBs. It is possible to upload such a large object file directly, but it could take a long time. If the network connection fails in the middle of the upload, we have to start over. A better solution is to slice a large object into smaller

parts and upload them independently. After all the parts are uploaded, the object store re-assembles the object from the parts. This process is called multipart upload.

Figure 9.25 illustrates how multipart upload works:

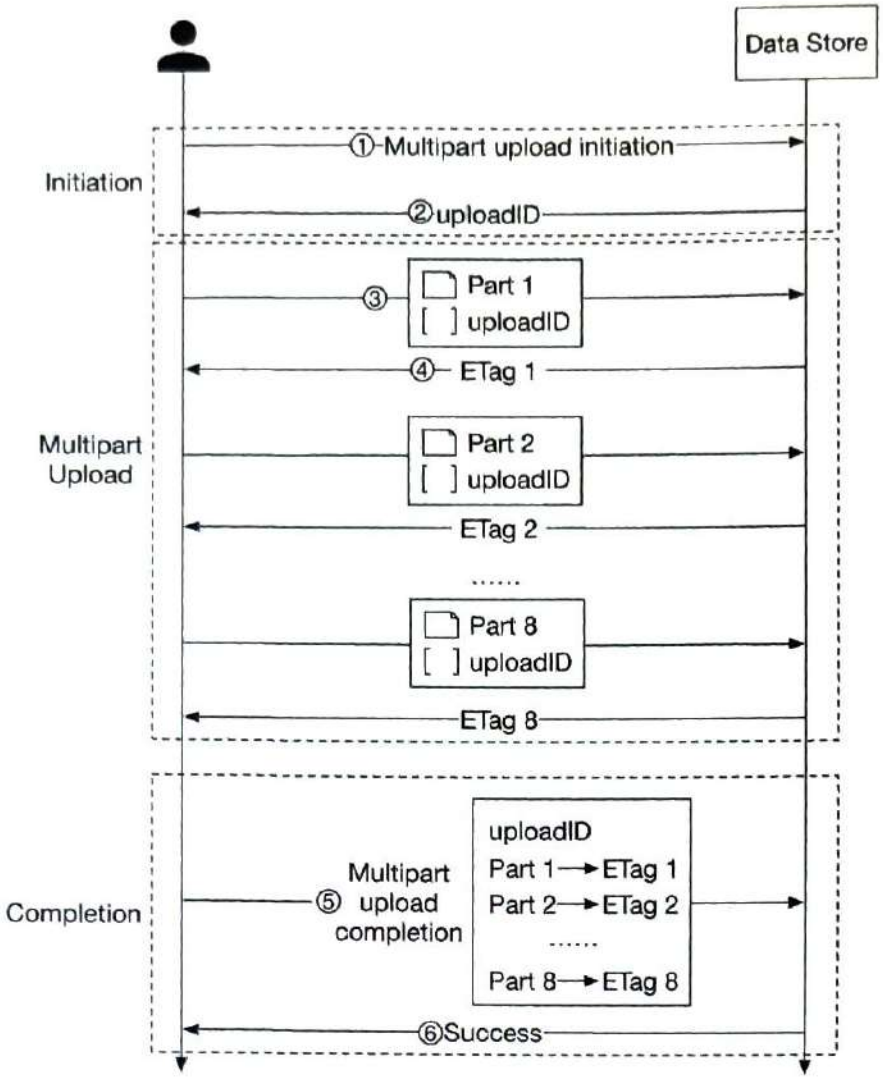


Figure 9.25: Multipart upload

1. The client calls the object storage to initiate a multipart upload.
2. The data store returns an `uploadID`, which uniquely identifies the upload.
3. The client splits the large file into small objects and starts uploading. Let's assume the size of the file is 1.6GB and the client splits it into 8 parts, so each part is 200MB in size. The client uploads the first part to the data store together with the uploadID it received in step 2.
4. When a part is uploaded, the data store returns an `ETag`, which is essentially the md5 checksum of that part. It is used to verify multipart uploads.
5. After all parts are uploaded, the client sends a complete multipart upload request, which includes the `uploadID`, part numbers, and `ETags`.
6. The data store reassembles the object from its parts based on the part number. Since

the object is really large, this process may take a few minutes. After reassembly is complete, it returns a success message to the client.

One potential problem with this approach is that old parts are no longer useful after the object has been reassembled from them. To solve this problem, we can introduce a garbage collection service responsible for freeing up space from parts that are no longer needed.

Garbage collection

Garbage collection is the process of automatically reclaiming storage space that is no longer used. There are a few ways that data might become garbage:

- Lazy object deletion. An object is marked as deleted at delete time without actually being deleted.
- Orphan data. For example, half uploaded data or abandoned multipart uploads.
- Corrupted data. Data that failed the checksum verification.

The garbage collector does not remove objects from the data store, right away. Deleted objects will be periodically cleaned up with a compaction mechanism.

The garbage collector is also responsible for reclaiming unused space in replicas. For replication, we delete the object from both primary and backup nodes. For erasure coding, if we use $(8 + 4)$ setup, we delete the object from all 12 nodes.

Figure 9.26 shows an example of how compaction works.

1. The garbage collector copies objects from `/data/b` to a new file named `/data/d`. Note the garbage collector skips "Object 2" and "Object 5" because the delete flag is set to true for both of them.
2. After all objects are copied, the garbage collector updates the `object_mapping` table. For example, the `obj_id` and `object_size` fields of "Object 3" remain the same, but `file_name` and `start_offset` are updated to reflect its new location. To ensure data consistency, it's a good idea to wrap the update operations to `file_name` and `start_offset` in a database transaction.

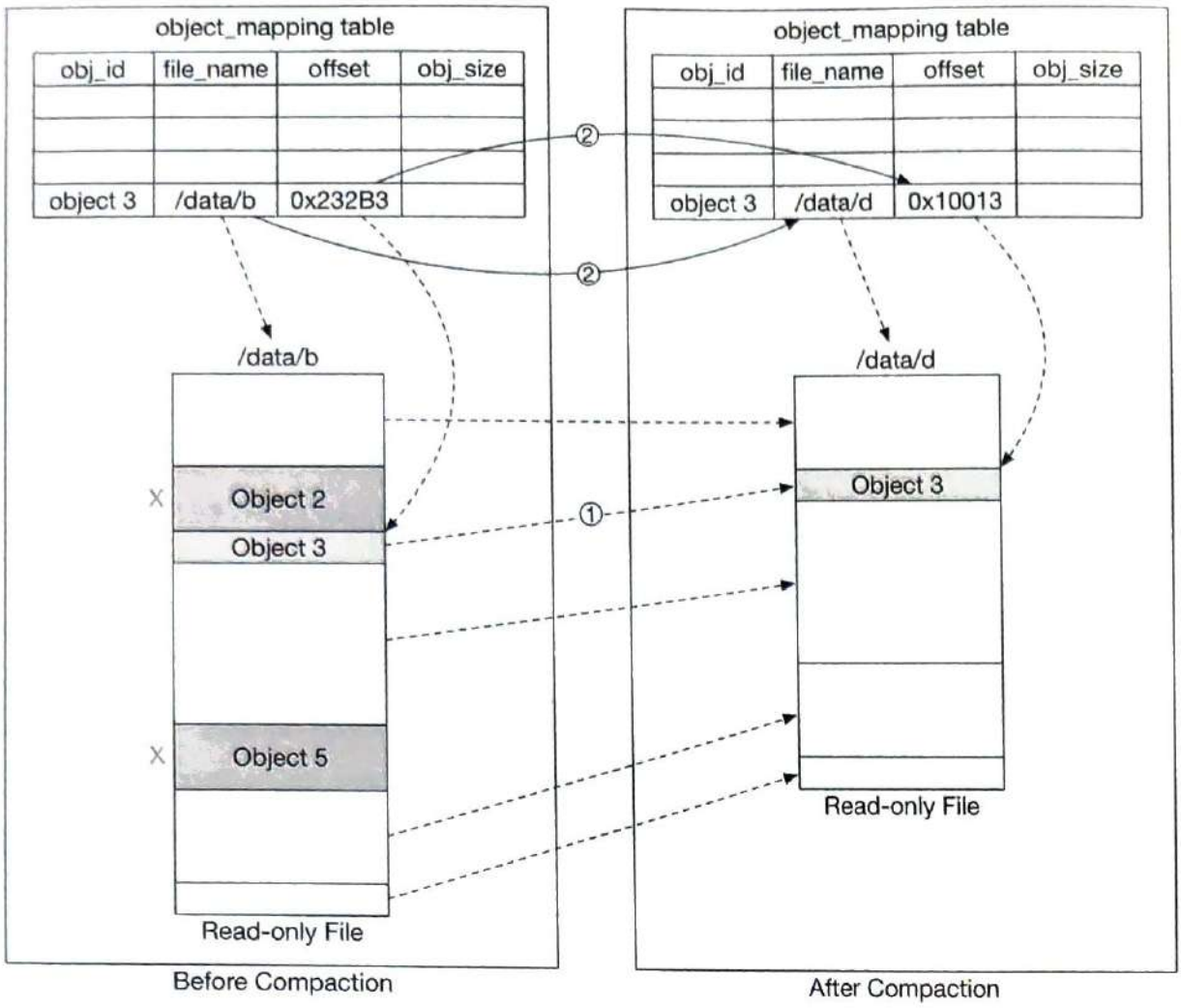


Figure 9.26: Compaction

As we can see from Figure 9.26, the size of the new file after compaction is smaller than the old file. To avoid creating a lot of small files, the garbage collector usually waits until there are a large number of read-only files to compact, and the compaction process appends objects from many read-only files into a few large new files.

Step 4 - Wrap Up

In this chapter, we described the high-level design for S3-like object storage. We compared the differences between block storage, file storage, and object storage.

The focus of this interview is on the design of object storage, so we listed how the uploading, downloading, listing objects in a bucket, and versioning of objects are typically done in object storage.

Then we dived deeper into the design. Object storage is composed of a data store and a metadata store. We explained how the data is persisted into the data store and discussed two methods for increasing reliability and durability: replication and erasure coding. For the metadata store, we explained how the multipart upload is executed and how to design the database schema to support typical use cases. Lastly, we explained how to shard the

Reference Material

[1] Fibre channel. https://en.wikipedia.org/wiki/Fibre_Channel.

[2] iSCSI. https://en.wikipedia.org/wiki/ISCSI.

[3] Server Message Block. https://en.wikipedia.org/wiki/Server_Message_Block.

[4] Network File System. https://en.wikipedia.org/wiki/Network_File_System.

[5] Amazon S3 Strong Consistency. https://aws.amazon.com/s3/consistency/.

[6] Serial Attached SCSI. https://en.wikipedia.org/wiki/Serial_Attached_SCSI.

[7] AWS CLI ls command. https://docs.aws.amazon.com/cli/latest/reference/s3/ls.html.

[8] Amazon S3 Service Level Agreement. https://aws.amazon.com/s3/sla/.

[9] Ambry. LinkedIn'sScalableGeo-DistributedObjectStore:https://assured-cloud-computing.illinois.edu/files/2014/03/Ambry-LinkedIns-Scalable-GeoDistributed-Object-Store.pdf.

[10] inode. https://en.wikipedia.org/wiki/Inode.

[11] Ceph's Rados Gateway. https://docs.ceph.com/en/pacific/radosgw/index.html.

[12] grpc. https://grpc.io/.

[13] Paxos. https://en.wikipedia.org/wiki/Paxos_(computer_science).

[14] Raft. https://raft.github.io/.

[15] Consistent hashing. https://www.toptal.com/big-data/consistent-hashing.

[16] RocksDB. https://github.com/facebook/rocksdb.

[17] SSTable. https://www.igvita.com/2012/02/06/sstable-and-log-structured-storage-leveldb/.

[18] B+ tree. https://en.wikipedia.org/wiki/B%2B_tree.

[19] SQLite. https://www.sqlite.org/index.html.

[20] Data Durability Calculation. https://www.backblaze.com/blog/cloud-storage-durability/.

[21] Rack. https://en.wikipedia.org/wiki/19-inch_rack.

[22] Erasure Coding. https://en.wikipedia.org/wiki/Erasure_code.

[23] Reed–Solomon error correction. https://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction.

[24] Erasure Coding Demystified. https://www.youtube.com/watch?v=Q5kVuM7zEUI.

[25] Checksum. https://en.wikipedia.org/wiki/Checksum.

[26] Md5. https://en.wikipedia.org/wiki/MD5.

[27] Sha1. https://en.wikipedia.org/wiki/SHA-1.

[28] Hmac. https://en.wikipedia.org/wiki/HMAC.

[29] TIMEUUID. https://docs.datastax.com/en/cql-oss/3.3/cql/cql_reference/timeuuid_functions_r.html.

10 Real-time Gaming Leaderboard

In this chapter, we are going to walk through the challenge of designing a leaderboard for an online mobile game.

What is a leaderboard? Leaderboards are common in gaming and elsewhere to show who is leading a particular tournament or competition. Users are assigned points for completing tasks or challenges, and whoever has the most points is at the top of the leaderboard. Figure 10.1 shows an example of a mobile game leaderboard. The leaderboard shows the ranking of the leading competitors and also displays the position of the user on it.

Figure 10.1: Leaderboard

Step 1 - Understand the Problem and Establish Design Scope

Leaderboards can be pretty straightforward, but there are a number of different matters that can add complexity. We should clarify the requirements.

Candidate: How is the score calculated for the leaderboard?
Interviewer: The user gets a point when they win a match. We can go with a simple point system in which each user has a score associated with them. Each time the user wins a match, we should add a point to their total score.

Candidate: Are all players included in the leaderboard?
Interviewer: Yes.

Candidate: Is there a time segment associated with the leaderboard?
Interviewer: Each month, a new tournament kicks off which starts a new leaderboard.

Candidate: Can we assume we only care about the top 10 users?
Interviewer: We want to display the top 10 users as well as the position of a specific user on the leaderboard. If time allows, let's also discuss how to return users who are four places above and below a specific user.

Candidate: How many users are in a tournament?
Interviewer: Average of 5 million daily active users (DAU) and 25 million monthly active users (MAU).

Candidate: How many matches are played on average during a tournament?
Interviewer: Each player plays 10 matches per day on average.

Candidate: How do we determine the rank if two players have the same score?
Interviewer: In this case, their ranks are the same. If time allows, we can talk about ways to break ties.

Candidate: Does the leaderboard need to be real-time?
Interviewer: Yes, we want to present real-time results, or as close as possible. It is not okay to present a batched history of results.

Now that we've gathered all the requirements, let's list the functional requirements.

- Display top 10 players on the leaderboard.
- Show a user's specific rank.
- Display players who are four places above and below the desired user (bonus).

Other than clarifying functional requirements, it's important to understand non-functional requirements.

Non-functional requirements

- Real-time update on scores.
- Score update is reflected on the leaderboard in real-time.
- General scalability, availability, and reliability requirements.

Back-of-the-envelope estimation

Let's take a look at some back-of-the-envelope calculations to determine the potential scale and challenges our solution will need to address.

With 5 million DAU, if the game had an even distribution of players during a 24-hour period, we would have an average of 50 users per second $\left(\frac{5{,}000{,}000 \text{ DAU}}{10^5 \text{ seconds}} =\sim 50\right)$. However, we know that usages most likely aren't evenly distributed, and potentially there are peaks during evenings when many people across different time zones have time to play. To account for this, we could assume that peak load would be 5 times the average.

Therefore we'd want to allow for a peak load of 250 users per second.

QPS for users scoring a point: if a user plays 10 games per day on average, the QPS for users scoring a point is: $50 \times 10 =\sim 500$. Peak QPS is 5x of the average: $500 \times 5 = 2{,}500$.

QPS for fetching the top 10 leaderboard: assume a user opens the game once a day and the top 10 leaderboard is loaded only when a user first opens the game. The QPS for this is around 50.

Step 2 - Propose High-level Design and Get Buy-in

In this section, we will discuss API design, high-level architecture, and data models.

API design

At a high level, we need the following three APIs:

`POST /v1/scores`

Update a user's position on the leaderboard when a user wins a game. The request parameters are listed below. This should be an internal API that can only be called by the game servers. The client should not be able to update the leaderboard score directly.

Field	Description
`user_id`	The user who wins a game.
`points`	The number of points a user gained by winning a game.

Table 10.1: Request parameters

Response:

Name	Description
`200 OK`	Successfully updated a user's score.
`400 Bad Request`	Failed to update a user's score.

Table 10.2: Response

`GET /v1/scores`

Fetch the top 10 players from the leaderboard.

Sample response:

```
{
  "data": [
  {
    "user_id": "user_id1",
    "user_name": "alice",
    "rank": 1,
    "score": 976
  },
  {
    "user_id": "user_id2",
    "user_name": "bob",
    "rank": 2,
    "score": 965
  }
  ],
  ...
  "total": 10
}
```

GET /v1/scores/{:user_id}

Fetch the rank of a specific user.

Field	Description
user_id	The ID of the user whose rank we would like to fetch.

Table 10.3: Request parameters

Sample response:

```
{
  "user_info": {
    "user_id": "user5",
    "score": 940,
    "rank": 6,
  }
}
```

High-level architecture

The high-level design diagram is shown in Figure 10.2. There are two services in this design. The game service allows users to play the game and the leaderboard service creates and displays a leaderboard.

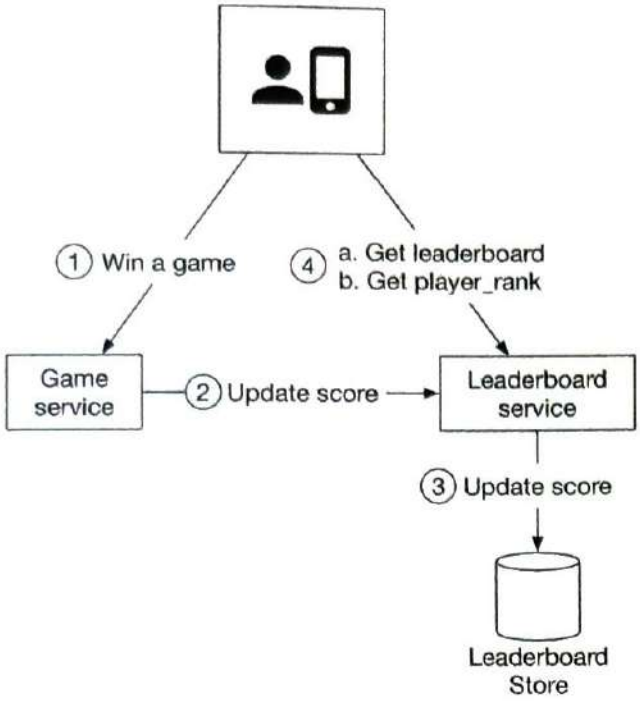


Figure 10.2: High-level design

1. When a player wins a game, the client sends a request to the game service.
2. The game service ensures the win is valid and calls the leaderboard service to update the score.
3. The leaderboard service updates the user's score in the leaderboard store.
4. A player makes a call to the leaderboard service directly to fetch leaderboard data, including:
   (a) top 10 leaderboard.
   (b) the rank of the player on the leaderboard.

Before settling on this design, we considered a few alternatives and decided against them. It might be helpful to go through the thought process of this and to compare different options.

Should the client talk to the leaderboard service directly?

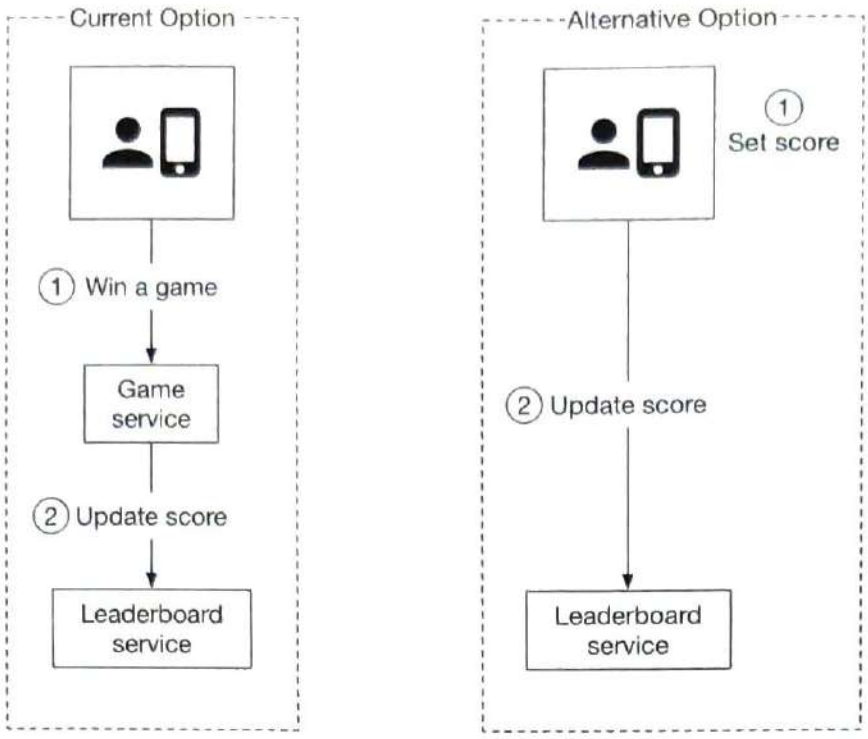


Figure 10.3: Who sets the leaderboard score

In the alternative design, the score is set by the client. This option is not secure because it is subject to man-in-the-middle attack [1], where players can put in a proxy and change scores at will. Therefore, we need the score to be set on the server-side.

Note that for server authoritative games such as online poker, the client may not need to call the game server explicitly to set scores. The game server handles all game logic, and it knows when the game finishes and could set the score without any client intervention.

Do we need a message queue between the game service and the leaderboard service?

The answer to this question highly depends on how the game scores are used. If the data is used in other places or supports multiple functionalities, then it might make sense to put data in Kafka as shown in Figure 10.4. This way, the same data can be consumed by multiple consumers, such as leaderboard service, analytics service, push notification service, etc. This is especially true when the game is a turn-based or multi-player game in which we need to notify other players about the score update. As this is not an explicit requirement based on the conversation with the interviewer, we do not use a message queue in our design.

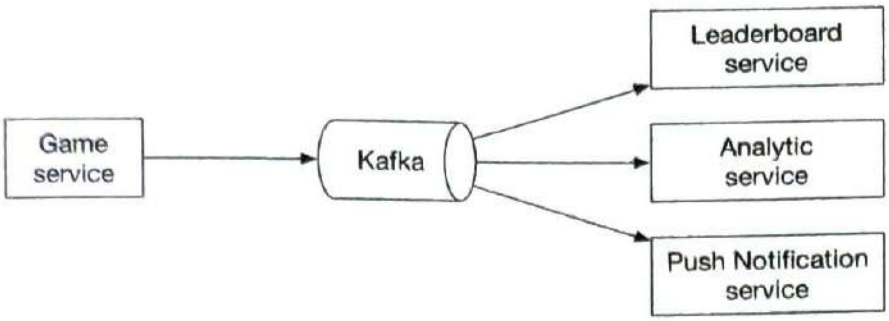


Figure 10.4: Game scores are used by multiple services

Data models

One of the key components in the system is the leaderboard store. We will discuss three potential solutions: relational database, Redis, and NoSQL (NoSQL solution is explained in deep dive section on page 309).

Relational database solution

First, let's take a step back and start with the simplest solution. What if the scale doesn't matter and we have only a few users?

We would most likely opt to have a simple leaderboard solution using a relational database system (RDS). Each monthly leaderboard could be represented as a database table containing user id and score columns. When the user wins a match, either award the user 1 point if they are new, or increase their existing score by 1 point. To determine a user's ranking on the leaderboard, we would sort the table by the score in descending order. The details are explained below.

Leaderboard DB table:

leaderboard	
user_id	varchar
score	int

Figure 10.5: Leaderboard table

In reality, the leaderboard table has additional information, such as a `game_id`, a timestamp, etc. However, the underlying logic of how to query and update the leaderboard remains the same. For simplicity, we assume only the current month's leaderboard data is stored in the leaderboard table.

A user wins a point:

Figure 10.6: A user wins a point

Assume every score update would be an increment of 1. If a user doesn't yet have an entry in the leaderboard for the month, the first insert would be:

```
INSERT INTO leaderboard (user_id, score) VALUES ('mary1934', 1)
  ;
```

An update to the user's score would be:

```
UPDATE leaderboard set score=score + 1 where user_id='mary1934';
```

Find a user's leaderboard position:

Figure 10.7: Find a user's leaderboard position

To fetch the user rank, we would sort the leaderboard table and rank by the score:

```
SELECT (@rownum := @rownum + 1) AS rank, user_id, score
FROM leaderboard
ORDER BY score DESC;
```

The result of the SQL query looks like this:

rank	user_id	score
1	happy_tomato	987
2	mallow	902
3	smith	870
4	mary1934	850

Table 10.4: Result sorted by score

This solution works when the data set is small, but the query becomes very slow when there are millions of rows. Let's take a look at why.

To figure out the rank of a user, we need to sort every single player into their correct spot on the leaderboard so we can determine exactly what the correct rank is. Remember that there can be duplicate scores as well, so the rank isn't just the position of the user in the list.

SQL databases are not performant when we have to process large amounts of continuously changing information. Attempting to do a rank operation over millions of rows is going to take 10s of seconds, which is not acceptable for the desired real-time approach. Since the data is constantly changing, it is also not feasible to consider a cache.

A relational database is not designed to handle the high load of read queries this implementation would require. An RDS could be used successfully if done as a batch operation, but that would not align with the requirement to return a real-time position for the user

on the leaderboard.

One optimization we can do is to add an index and limit the number of pages to scan with the LIMIT clause. The query looks like this:

```
SELECT (@rownum := @rownum + 1) AS rank, user_id, score
FROM leaderboard
ORDER BY score DESC
LIMIT 10
```

However, this approach doesn't scale well. First, finding a user's rank is not performant because it essentially requires a table scan to determine the rank. Second, this approach doesn't provide a straightforward solution for determining the rank of a user who is not at the top of the leaderboard.

Redis solution

We want to find a solution that gives us predictable performance even for millions of users and allows us to have easy access to common leaderboard operations, without needing to fall back on complex DB queries.

Redis provides a potential solution to our problem. Redis is an in-memory data store supporting key-value pairs. Since it works in memory, it allows for fast reads and writes. Redis has a specific data type called **sorted sets** that are ideal for solving leaderboard system design problems.

What are sorted sets?

A sorted set is a data type similar to a set. Each member of a sorted set is associated with a score. The members of a set must be unique, but scores may repeat. The score is used to rank the sorted set in ascending order.

Our leaderboard use case maps perfectly to sorted sets. Internally, a sorted set is implemented by two data structures: a hash table and a skip list [2]. The hash table maps users to scores and the skip list maps scores to users. In sorted sets, users are sorted by scores. A good way to understand a sorted set is to picture it as a table with score and member columns as shown in Figure 10.8. The table is sorted by score in descending order.

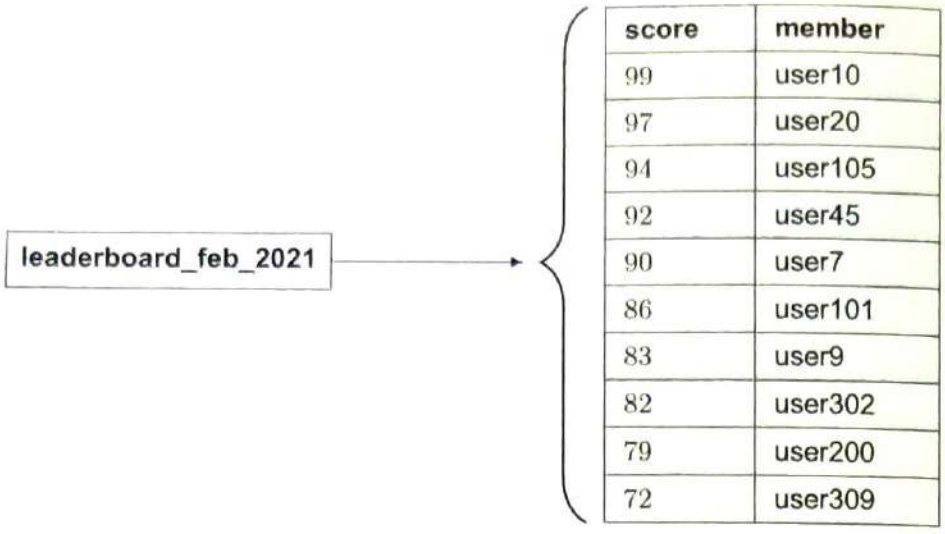


Figure 10.8: February leaderboard is represented by the sorted set

In this chapter, we don't go into the full detail of the sorted set implementation, but we do go over the high-level ideas.

A skip list is a list structure that allows for fast search. It consists of a base sorted linked list and multi-level indexes. Let's take a look at an example. In Figure 10.9, the base list is a sorted singly-linked list. The time complexity of insertion, removal, and search operations is $O(n)$.

How can we make those operations faster? One idea is to get to the middle quickly, as the binary search algorithm does. To achieve that, we add a level 1 index that skips every other node, and then a level 2 index that skips every other node of the level 1 indexes. We keep introducing additional levels, with each new level skipping every other nodes of the previous level. We stop this addition when the distance between nodes is $\frac{n}{2} - 1$, where n is the total number of nodes. As shown in Figure 10.9, searching for number 45 is a lot faster when we have multi-level indexes.

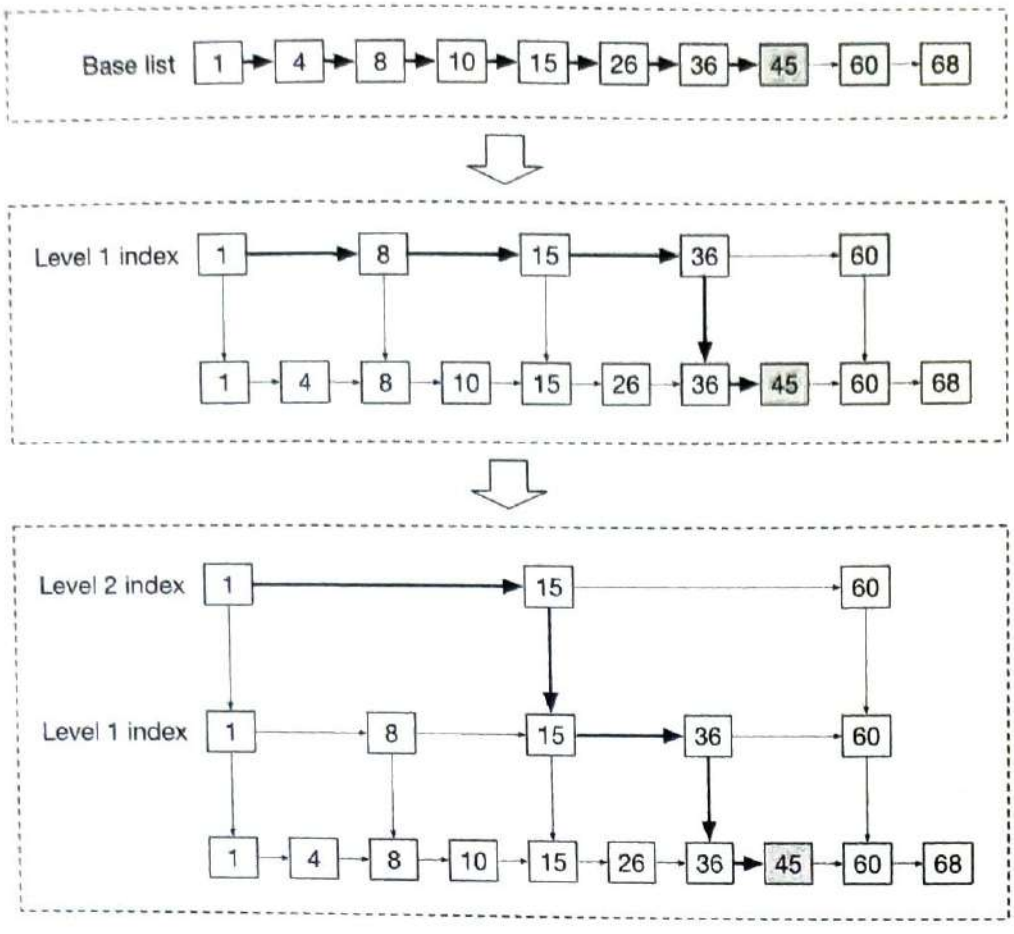


Figure 10.9: Skip list

When the data set is small, the speed improvement using the skip list isn't obvious. Figure 10.10 shows an example of a skip list with 5 levels of indexes. In the base linked list, it needs to travel 62 nodes to reach the correct node. In the skip list, it only needs to traverse 11 nodes [3].

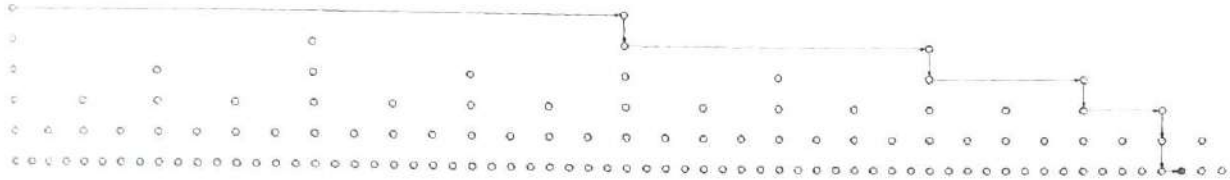


Figure 10.10: Skip list with 5 levels of indexes

Sorted sets are more performant than a relational database because each element is automatically positioned in the right order during insert or update, as well as the fact that the complexity of an add or find operation in a sorted set is logarithmic: $O(\log(n))$.

In contrast, to calculate the rank of a specific user in a relational database, we need to run nested queries:

```
SELECT *,(SELECT COUNT(*) FROM leaderboard lb2
WHERE lb2.score >= lb1.score) RANK
FROM leaderboard lb1
WHERE lb1.user_id = {:user_id};
```

Implementation using Redis sorted sets

Now that we know sorted sets are fast, let's take a look at the Redis operations we will use to build our leaderboard [4] [5] [6] [7]:

- `ZADD`: insert the user into the set if they don't yet exist. Otherwise, update the score for the user. It takes $O(\log(n))$ to execute.
- `ZINCRBY`: increment the score of the user by the specified increment. If the user doesn't exist in the set, then it assumes the score starts at 0. It takes $O(\log(n))$ to execute.
- `ZRANGE/ZREVRANGE`: fetch a range of users sorted by the score. We can specify the order (range vs. revrange), the number of entries, and the position to start from. This takes $O(\log(n)+m)$ to execute, where m is the number of entries to fetch (which is usually small in our case), and n is the number of entries in the sorted set.
- `ZRANK/ZREVRANK`: fetch the position of any user sorting in ascending/descending order in logarithmic time.

Workflow with sorted sets

1. A user scores a point

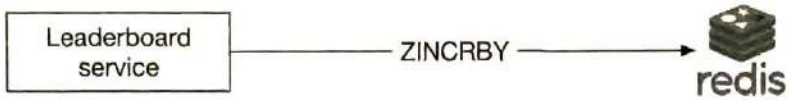


Figure 10.11: A user scores a point

Every month we create a new leaderboard sorted set and the previous ones are moved to historical data storage. When a user wins a match, they score 1 point; so we call `ZINCRBY` to increment the user's score by 1 in that month's leaderboard, or add the user to the leaderboard set if they weren't already there. The syntax for `ZINCRBY` is:

```
ZINCRBY <key> <increment> <user>
```

The following command adds a point to user `mary1934` after they win a match.

```
ZINCRBY leaderboard_feb_2021 1 'mary1934'
```

2. A user fetches the top 10 global leaderboard

Figure 10.12: Fetch top 10 global leaderboard

We will call ZREVRANGE to obtain the members in descending order because we want the highest scores, and pass the WITHSCORES attribute to ensure that it also returns the total score for each user, as well as the set of users with the highest scores. The following command fetches the top 10 players on the Feb-2021 leaderboard.

```
ZREVRANGE leaderboard_feb_2021 0 9 WITHSCORES
```

This returns a list like this:

```
[(user2,score2),(user1,score1),(user5,score5)...]
```

3. A user wants to fetch their leaderboard position

Figure 10.13: Fetch a user's leaderboard position

To fetch the position of a user in the leaderboard, we will call ZREVRANK to retrieve their rank on the leaderboard. Again, we call the rev version of the command because we want to rank scores from high to low.

```
ZREVRANK leaderboard_feb_2021 'mary1934'
```

4. Fetch the relative position in the leaderboard for a user. An example is shown in Figure 10.14.

Rank	Player	Points
267	Aquaboys	876
258	B team	845
259	Berlin's Angels	832
360	GrendelTeam	799
361	Mallow007	785
362	Woo78	743
363	milan~114	732
364	G3^^^^2	726
365	Mailso_91_	712

Figure 10.14: Fetch 4 players above and below

While not an explicit requirement, we can easily fetch the relative position for a user by leveraging `ZREVRANGE` with the number of results above and below the desired player. For example, if user `Mallow007`'s rank is 361 and we want to fetch 4 players above and below them, we would run the following command.

```
ZREVRANGE leaderboard_feb_2021 357 365
```

Storage requirement

At a minimum, we need to store the user id and score. The worst-case scenario is that all 25 million monthly active users have won at least one game, and they all have entries in the leaderboard for the month. Assuming the id is a 24-character string and the score is a 16-bit integer (or 2 bytes), we need 26 bytes of storage per leaderboard entry. Given the worst-case scenario of one leaderboard entry per MAU, we would need 26 bytes $\times 25$ million $= 650$ million bytes or $\sim$ 650MB for leaderboard storage in the Redis cache. Even if we double the memory usage to account for the overhead of the skip list and the hash for the sorted set, one modern Redis server is more than enough to hold the data.

Another related factor to consider is CPU and I/O usage. Our peak QPS from the back-of-the-envelope estimation is 2500 updates/sec. This is well within the performance envelope of a single Redis server.

One concern about the Redis cache is persistence, as a Redis node might fail. Luckily, Redis does support persistence, but restarting a large Redis instance from disk is slow. Usually, Redis is configured with a read replica, and when the main instance fails, the read replica is promoted, and a new read replica is attached.

Besides, we need to have 2 supporting tables (user and point) in a relational database like MySQL. The user table would store the user ID and user's display name (in a real-world application, this would contain a lot more data). The point table would contain the user id, score, and timestamp when they won a game. This can be leveraged for other game functions such as play history, and can also be used to recreate the Redis leaderboard in the event of an infrastructure failure.

As a small performance optimization, it may make sense to create an additional cache of the user details, potentially for the top 10 players since they are retrieved most frequently. However, this doesn't amount to a large amount of data.

Step 3 - Design Deep Dive

Now that we've discussed the high-level design, let's dive into the following:

- Whether or not to use a cloud provider
  - Manage our own services
  - Leverage cloud service providers like Amazon Web Services (AWS)
- Scaling Redis
- Alternative solution: NoSQL
- Other considerations

To use a cloud provider or not

Depending on the existing infrastructure, we generally have two options for deploying our solution. Let's take a look at each of them.

Manage our own services

In this approach, we will create a leaderboard sorted set each month to store the leaderboard data for that period. The sorted set stores member and score information. The rest of the details about the user, such as their name and profile image, are stored in MySQL databases. When fetching the leaderboard, besides the leaderboard data, API servers also query the database to fetch corresponding users' names and profile images to display on the leaderboard. If this becomes too inefficient in the long term, we can leverage a user profile cache to store users' details for the top 10 players. The design is shown in Figure 10.15.

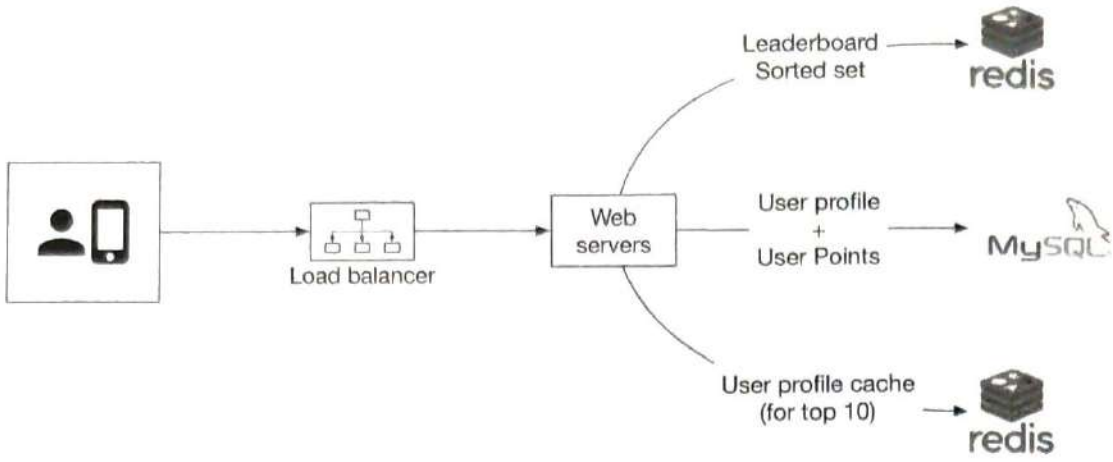


Figure 10.15: Manage our own services

Build on the cloud

The second approach is to leverage cloud infrastructures. In this section, we assume our existing infrastructure is built on AWS and that it's a natural fit to build the leaderboard on the cloud. We will use two major AWS technologies in this design: Amazon API Gateway and AWS Lambda function [8]. The Amazon API gateway provides a way to define the HTTP endpoints of a RESTful API and connect it to any backend services. We use it to connect to our AWS lambda functions. The mapping between Restful APIs and Lambda functions is shown in Table 10.5.

APIs	Lambda function
`GET /v1/scores`	`LeaderboardFetchTop10`
`GET /v1/scores/{:user_id}`	`LeaderboardFetchPlayerRank`
`POST /v1/scores`	`LeaderboardUpdateScore`

Table 10.5: Lambda functions

AWS Lambda is one of the most popular serverless computing platforms. It allows us to run code without having to provision or manage the servers ourselves. It runs only when needed and will scale automatically based on traffic. Serverless is one of the hottest topics in cloud services and is supported by all major cloud service providers. For example, Google Cloud has Google Cloud Functions [9] and Microsoft has named its offering Microsoft Azure Functions [10].

At a high level, our game calls the Amazon API Gateway, which in turn invokes the appropriate lambda functions. We will use AWS Lambda functions to invoke the appropriate commands on the storage layer (both Redis and MySQL), return the results back to the API Gateway, and then to the application.

We can leverage Lambda functions to perform the queries we need without having to spin up a server instance. AWS provides support for Redis clients that can be called from the Lambda functions. This also allows for auto-scaling as needed with DAU growth. Design diagrams for a user scoring a point and retrieving the leaderboard are shown below:

Use case 1: scoring a point

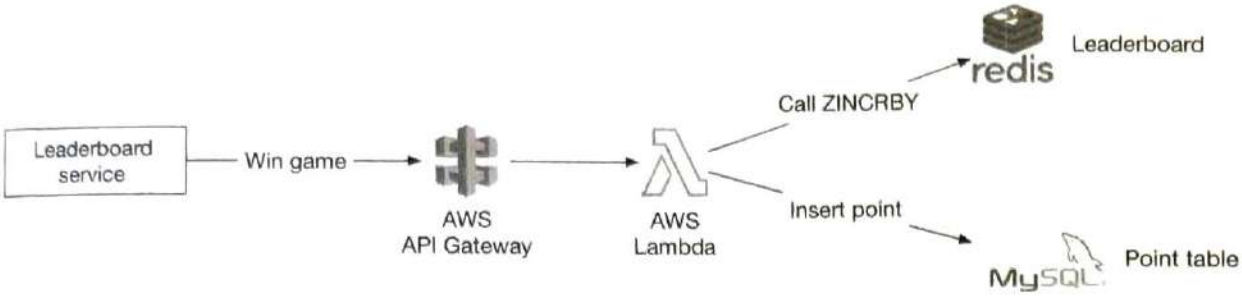


Figure 10.16: Score a point

Use case 2: retrieving leaderboard

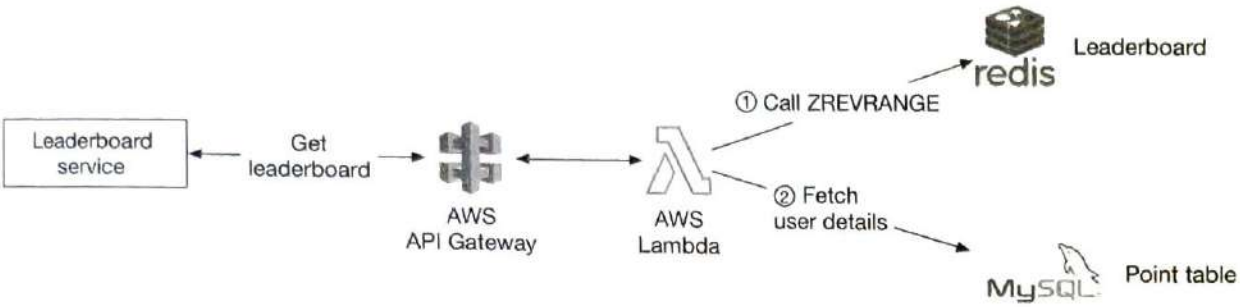


Figure 10.17: Retrieve leaderboard

Lambdas are great because they are a serverless approach, and the infrastructure will take care of auto-scaling the function as needed. This means we don't need to manage the scaling and environment setup and maintenance. Given this, we recommend going with a serverless approach if we build the game from the ground up.

Scaling Redis

With 5 million DAU, we can get away with one Redis cache from both a storage and QPS perspective. However, let's imagine we have 500 million DAU, which is 100 times our original scale. Now our worst-case scenario for the size of the leaderboard goes up to 65GB (650MB $\times$ 100), and our QPS goes up to 250,000 (2,500 $\times$ 100) queries per second. This calls for a sharding solution.

Data sharding

We consider sharding in one of the following two ways: fixed or hash partitions.

Fixed partition

One way to understand fixed partitions is to look at the overall range of points on the leaderboard. Let's say that the number of points won in one month ranges from 1 to 1000, and we break up the data by range. For example, we could have 10 shards and each shard would have a range of 100 scores (For example, $1 \sim 100$, $101 \sim 200$, $201 \sim 300$, ...) as shown in Figure 10.18.

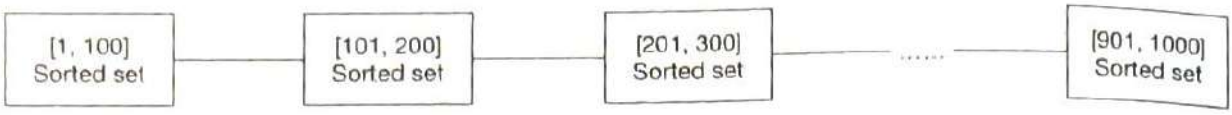


Figure 10.18: Fixed partition

For this to work, we want to ensure there is an even distribution of scores across the leaderboard. Otherwise, we need to adjust the score range in each shard to make sure of a relatively even distribution. In this approach, we shard the data ourselves in the application code.

When we are inserting or updating the score for a user, we need to know which shard they are in. We could do this by calculating the user's current score from the MySQL database. This can work, but a more performant option is to create a secondary cache to store the mapping from user ID to score. We need to be careful when a user increases their score and moves between shards. In this case, we need to remove the user from their current shard and move them to the new shard.

To fetch the top 10 players in the leaderboard, we would fetch the top 10 players from the shard (sorted set) with the highest scores. In Figure 10.18, the last shard with scores [901, 1000] contains the top 10 players.

To fetch the rank of a user, we would need to calculate the rank within their current shard (local rank), as well as the total number of players with higher scores in all of the shards. Note that the total number of players in a shard can be retrieved by running the `info keyspace` command in $O(1)$ [11].

Hash partition

A second approach is to use the Redis cluster, which is desirable if the scores are very clustered or clumped. Redis cluster provides a way to shard data automatically across multiple Redis nodes. It doesn't use consistent hashing but a different form of sharding, where every key is part of a **hash slot**. There are 16384 hash slots [12] and we can compute the hash slot of a given key by doing `CRC16(key) %16384` [13]. This allows us to add and remove nodes in the cluster easily without redistributing all the keys. In Figure 10.19, we have 3 nodes, where:

- The first node contains hash slots [0, 5500].
- The second node contains hash slots [5501, 11000].
- The third node contains hash slots [11001, 16383].

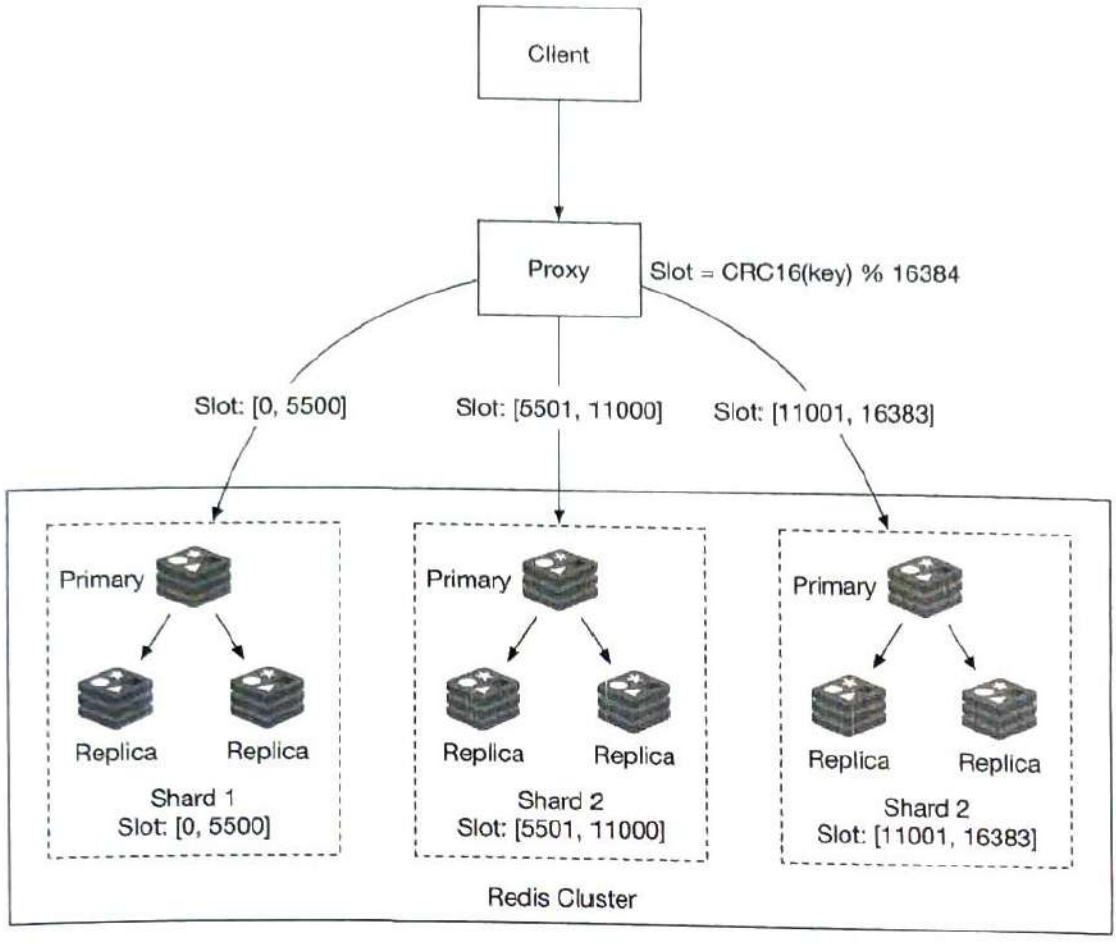


Figure 10.19: Hash partition

An update would simply change the score of the user in the corresponding shard (determined by `CRC16(key) %16384`). Retrieving the top 10 players on the leaderboard is more complicated. We need to gather the top 10 players from each shard and have the application sort the data. A concrete example is shown in Figure 10.20. Those queries can be parallelized to reduce latency.

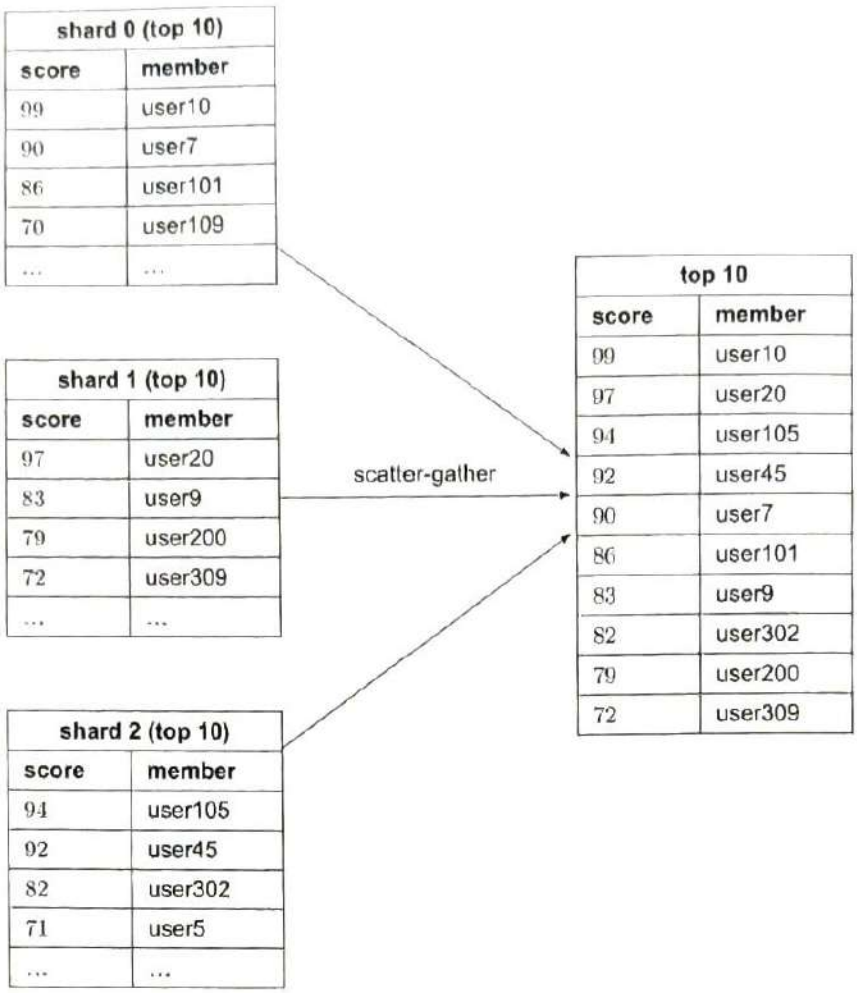


Figure 10.20: Scatter-gather

This approach has a few limitations:

- When we need to return top k results (where k is a very large number) on the leaderboard, the latency is high because a lot of entries are returned from each shard and need to be sorted.
- Latency is high if we have lots of partitions because the query has to wait for the slowest partition.
- Another issue with this approach is that it doesn't provide a straightforward solution for determining the rank of a specific user.

Therefore, we lean towards the first proposal: fixed partition.

Sizing a Redis node

There are multiple things to consider when sizing the Redis nodes [14]. Write-heavy applications require much more available memory, since we need to be able to accommodate all of the writes to create the snapshot in case of a failure. To be safe, allocate twice the amount of memory for write-heavy applications.

Redis provides a tool called Redis-benchmark that allows us to benchmark the perfor-

mance of the Redis setup, by simulating multiple clients executing multiple queries and returning the number of requests per second for the given hardware. To learn more about Redis-benchmark, see [15].

Alternative solution: NoSQL

An alternative solution to consider is NoSQL databases. What kind of NoSQL should we use? Ideally, we want to choose a NoSQL that has the following properties:

- Optimized for writes.
- Efficiently sort items within the same partition by score.

NoSQL databases such as Amazon's DynamoDB [16], Cassandra, or MongoDB can be a good fit. In this chapter, we use DynamoDB as an example. DynamoDB is a fully managed NoSQL database that offers reliable performance and great scalability. To allow efficient access to data with attributes other than the primary key, we can leverage global secondary indexes [17] in DynamoDB. A global secondary index contains a selection of attributes from the parent table, but they are organized using a different primary key. Let's take a look at an example.

The updated system diagram is shown in Figure 10.21. Redis and MySQL are replaced with DynamoDB.

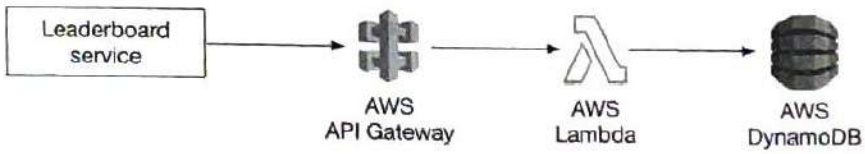


Figure 10.21: DynamoDB solution

Assume we design the leaderboard for a chess game and our initial table is shown in Figure 10.22. It is a denormalized view of the leaderboard and user tables and contains everything needed to render a leaderboard.

Primary key	Attributes			
user_id	score	email	profile_pic	leaderboard_name
lovelove	309	love@test.com	https://cdn.example/3.png	chess#2020-02
i_love_tofu	209	test@test.com	https://cdn.example/p.png	chess#2020-02
golden_gate	103	gold@test.com	https://cdn.example/2.png	chess#2020-03
pizza_or_bread	203	piz@test.com	https://cdn.example/31.png	chess#2021-05
ocean	10	oce@test.com	https://cdn.example/32.png	chess#2020-02
...	...	...	...	...

Figure 10.22: Denormalized view of the leaderboard and user tables

This table scheme works, but it doesn't scale well. As more rows are added, we have to scan the entire table to find the top scores.

linear scan, we need to add indexes. Our first attempt is to use year-month} as the partition key and the score as the sort key, as gure 10.23.

condary Index		Attributes		
ey	Sort key (score)	user_id	email	profile_pic
-02	309	lovelove	love@test.com	https://cdn.example/3.png
02	209	i_love_tofu	test@test.com	https://cdn.example/p.png
03	103	golden_gate	gold@test.com	https://cdn.example/2.png
02	203	pizza_or_bread	piz@test.com	https://cdn.example/31.png
02	10	ocean	oce@test.com	https://cdn.example/32.png
	...	...	...	...

Figure 10.23: Partition key and sort key

t runs into issues at a high load. DynamoDB splits data across multiple sistent hashing. Each item lives in a corresponding node based on its 're want to structure the data so that data is evenly distributed across r table design (Figure 10.23), all the data for the most recent month n one partition and that partition becomes a hot partition. How can we n?

ta into n partitions and append a partition number (`user_id % ions`) to the partition key. This pattern is called write sharding. Write nplexity for both read and write operations, so we should consider the y.

ion we need to answer is, how many partitions should we have? It rite volume or DAU. The important thing to remember is that there is n load on partitions and read complexity. Because data for the same venly across multiple partitions, the load for a single partition is much to read items for a given month, we have to query all the partitions ults, which adds read complexity.

oks something like this: `game_name#{year-month}#p{partition_number}`. the updated schema table.

Global Secondary Index	
Partition key (PK)	Sort key (score)
ches#2020-02#p0	309
ches#2020-02#p1	209
ches#2020-03#p2	103
ches#2020-02#p1	203
ches#2020-02#p2	10
	...

Figu

global secondary index us
tition key and the score as th
sorted within their own par
in order to fetch the top 10
ter" mentioned earlier. We
is the "scatter" portion), and
ing all the partitions (this is
).

top 10 from partition 0 (scatt

rtition key (PK)	Sort key (score)	use
ches#2020-02#p0	309	lovelo
	...	...

top 10 from partition 1 (scatter)

rtition key (PK)	Sort key (score)	user_id
ches#2020-02#p1	209	i_love_t
ches#2020-02#p1	203	pizza_or
	...	...

top 10 from partition 2 (scatter)

rtition key (PK)	Sort key (score)	user_id
ches#2020-02#p2	10	ocean
	...	...

Figure

Global Secondary Index		Attributes		
Partition key (PK)	**Sort key (score)**	**user_id**	**email**	**profile_pic**
chess#2020-02#p0	309	lovelove	love@test.com	https://cdn.example/3.png
chess#2020-02#p1	209	i_love_tofu	test@test.com	https://cdn.example/p.png
chess#2020-03#p2	103	golden_gate	gold@test.com	https://cdn.example/2.png
chess#2020-02#p1	203	pizza_or_bread	piz@test.com	https://cdn.example/31.png
chess#2020-02#p2	10	ocean	oce@test.com	https://cdn.example/32.png
...	...	...	...	...

Figure 10.24: Updated partition key

The global secondary index uses `game_name#{year-month}#p{partition_number}` as the partition key and the score as the sort key. What we end up with are n partitions that are all sorted within their own partition (locally sorted). If we assume we had 3 partitions, then in order to fetch the top 10 leaderboard, we would use the approach called "scatter-gather" mentioned earlier. We would fetch the top 10 results in each of the partitions (this is the "scatter" portion), and then we would allow the application to sort the results among all the partitions (this is the "gather" portion). An example is shown in Figure 10.25.

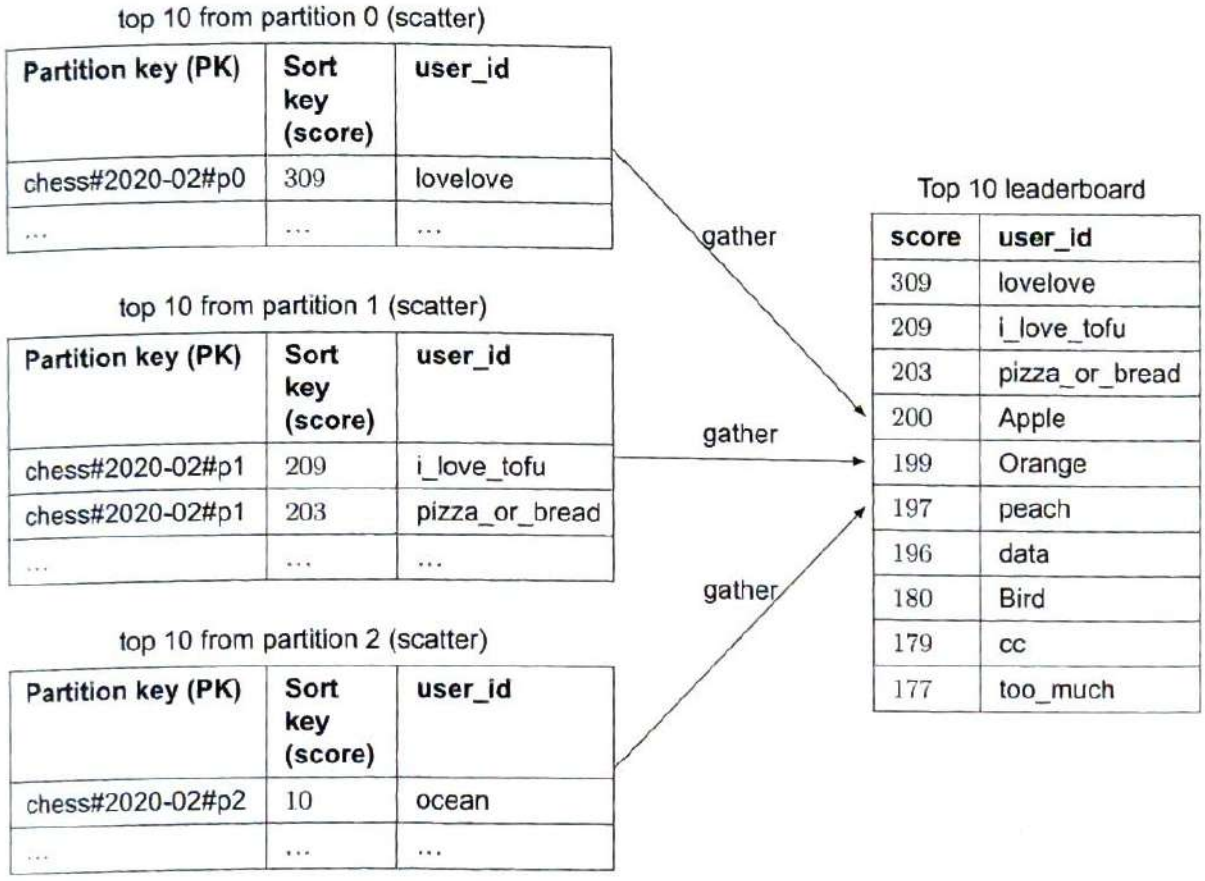


Figure 10.25: Scatter-gather

How do we decide on the number of partitions? This might require some careful benchmarking. More partitions decrease the load on each partition but add complexity, as we need to scatter across more partitions to build the final leaderboard. By employing benchmarking, we can see the trade-off more clearly.

However, similar to the Redis partition solution mentioned earlier, this approach doesn't provide a straightforward solution for determining the relative rank of a user. But it is possible to get the percentile of a user's position, which could be good enough. In real life, telling a player that they are in the top $10 \sim 20\%$ might be better than showing the exact rank at eg. 1,200,001. Therefore, if the scale is large enough that we needed to shard, we could assume that the score distributions are roughly the same across all shards. If this assumption is true, we could have a cron job that analyzes the distribution of the score for each shard, and caches that result.

The result would look something like this:

$$\begin{aligned} &\text{10th percentile} = \text{score} < 100 \\ &\text{20th percentile} = \text{score} < 500 \\ &\quad\vdots \qquad\qquad\quad \vdots \\ &\text{90th percentile} = \text{score} < 6500 \end{aligned}$$

Then we could quickly return a user's relative ranking (say 90th percentile).

Step 4 - Wrap Up

In this chapter, we have created a solution for building a real-time game leaderboard with the scale of millions of DAU. We explored the straightforward solution of using a MySQL database and rejected that approach because it does not scale to millions of users. We then designed the leaderboard using Redis sorted sets. We also looked into scaling the solution to 500 million DAU, by leveraging sharding across different Redis caches. We also proposed an alternative NoSQL solution.

In the event you have some extra time at the end of the interview, you can cover a few more topics:

Faster retrieval and breaking tie

A Redis Hash provides a map between string fields and values. We could leverage a hash for 2 use cases:

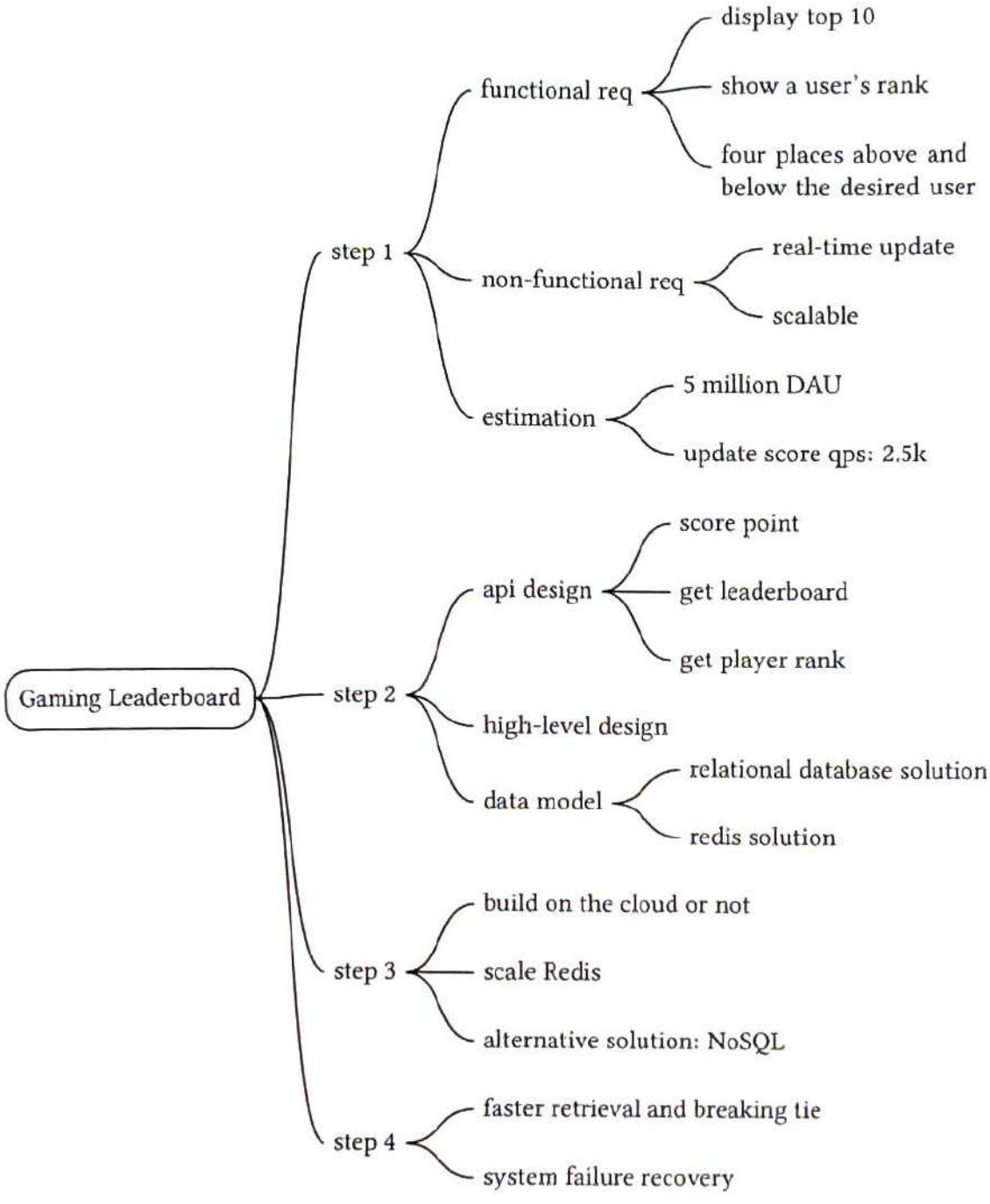
1. To store a map of the user id to the user object that we can display on the leaderboard. This allows for faster retrieval than having to go to the database to fetch the user object.
2. In the case of two players having the same scores, we could rank the users based on who received that score first. When we increment the score of the user, we can also store a map of the user id to the timestamp of the most recently won game. In the case of a tie, the user with the older timestamp ranks higher.

System failure recovery

The Redis cluster can potentially experience a large-scale failure. Given the design above, we could create a script that leverages the fact that the MySQL database records an entry with a timestamp each time a user won a game. We could iterate through all of the entries for each user, and call `ZINCRBY` once per entry, per user. This would allow us to recreate the leaderboard offline if necessary, in case of a large-scale outage.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Chapter Summary



Reference Material

[1] Man-in-the-middle attack. https://en.wikipedia.org/wiki/Man-in-the-middle_attack.

[2] Redis Sorted Set source code. https://github.com/redis/redis/blob/unstable/src/t_zset.c.

[3] Geekbang. https://static001.geekbang.org/resource/image/46/a9/46d283cd82c987153b3fe0c76dfba8a9.jpg.

[4] Building real-time Leaderboard with Redis. https://medium.com/@sandeep4.verma/building-real-time-leaderboard-with-redis-82c98aa47b9f.

[5] Build a real-time gaming leaderboard with Amazon ElastiCache for Redis. https://aws.amazon.com/blogs/database/building-a-real-time-gaming-leaderboard-with-amazon-elasticache-for-redis.

[6] How we created a real-time Leaderboard for a million Users. https://levelup.gitconnected.com/how-we-created-a-real-time-leaderboard-for-a-million-users-555aaa3ccf7b.

[7] Leaderboards. https://redislabs.com/solutions/use-cases/leaderboards/.

[8] Lambda. https://aws.amazon.com/lambda/.

[9] Google Cloud Functions. https://cloud.google.com/functions.

[10] Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[11] Info command. https://redis.io/commands/INFO.

[12] Why redis cluster only have 16384 slots. https://stackoverflow.com/questions/36203532/why-redis-cluster-only-have-16384-slots.

[13] Cyclic redundancy check. https://en.wikipedia.org/wiki/Cyclic_redundancy_check.

[14] Choosing your node size. https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/nodes-select-size.html.

[15] How fast is Redis? https://redis.io/topics/benchmarks.

[16] Using Global Secondary Indexes in DynamoDB. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html.

[17] Leaderboard & Write Sharding. https://www.dynamodbguide.com/leaderboard-write-sharding/.

11 Payment System

In this chapter, we design a payment system. E-commerce has exploded in popularity across the world in recent years. What makes every transaction possible is a payment system running behind the scenes. A reliable, scalable, and flexible payment system is essential.

What is a payment system? According to Wikipedia, "a payment system is any system used to settle financial transactions through the transfer of monetary value. This includes the institutions, instruments, people, rules, procedures, standards, and technologies that make its exchange possible" [1].

A payment system is easy to understand on the surface but is also intimidating for many developers to work on. A small slip could potentially cause significant revenue loss and destroy credibility among users. But fear not! In this chapter, we demystify payment systems.

Step 1 - Understand the Problem and Establish Design Scope

A payment system can mean very different things to different people. Some may think it's a digital wallet like Apple Pay or Google Pay. Others may think it's a backend system that handles payments such as PayPal or Stripe. It is very important to determine the exact requirements at the beginning of the interview. These are some questions you can ask the interviewer:

Candidate: What kind of payment system are we building?
Interviewer: Assume you are building a payment backend for an e-commerce application like Amazon.com. When a customer places an order on Amazon.com, the payment system handles everything related to money movement.

Candidate: What payment options are supported? Credit cards, PayPal, bank cards, etc?
Interviewer: The payment system should support all of these options in real life. However, in this interview, we can use credit card payment as an example.

Candidate: Do we handle credit card payment processing ourselves?

Interviewer: No, we use third-party payment processors, such as Stripe, Braintree, Square, etc.

Candidate: Do we store credit card data in our system?
Interviewer: Due to extremely high security and compliance requirements, we do not store card numbers directly in our system. We rely on third-party payment processors to handle sensitive credit card data.

Candidate: Is the application global? Do we need to support different currencies and international payments?
Interviewer: Great question. Yes, the application would be global but we assume only one currency is used in this interview.

Candidate: How many payment transactions per day?
Interviewer: 1 million transactions per day.

Candidate: Do we need to support the pay-out flow, which an e-commerce site like Amazon uses to pay sellers every month?
Interviewer: Yes, we need to support that.

Candidate: I think I have gathered all the requirements. Is there anything else I should pay attention to?
Interviewer: Yes. A payment system interacts with a lot of internal services (accounting, analytics, etc.) and external services (payment service providers). When a service fails, we may see inconsistent states among services. Therefore, we need to perform reconciliation and fix any inconsistencies. This is also a requirement.

With these questions, we get a clear picture of both the functional and non-functional requirements. In this interview, we focus on designing a payment system that supports the following.

Functional requirements

- Pay-in flow: payment system receives money from customers on behalf of sellers.
- Pay-out flow: payment system sends money to sellers around the world.

Non-functional requirements

- Reliability and fault tolerance. Failed payments need to be carefully handled.
- A reconciliation process between internal services (payment systems, accounting systems) and external services (payment service providers) is required. The process asynchronously verifies that the payment information across these systems is consistent.

Back-of-the-envelope estimation

The system needs to process 1 million transactions per day, which is 1,000,000 transactions $/10^5$ seconds = 10 transactions per second (TPS). 10 TPS is not a big number for a typical database, which means the focus of this system design interview is on how to

correctly handle payment transactions, rather than aiming for high throughput.

Step 2 - Propose High-level Design and Get Buy-in

At a high level, the payment flow is broken down into two steps to reflect how flows:

- Pay-in flow
- Pay-out flow

Take the e-commerce site, Amazon, as an example. After a buyer places a money flows into Amazon's bank account, which is the pay-in flow. Althoug is in Amazon's bank account, Amazon does not own all of the money. Th a substantial part of it and Amazon only works as the money custodian fo when the products are delivered and money is released, the balance after fe from Amazon's bank account to the seller's bank account. This is the pay simplified pay-in and pay-out flows are shown in Figure 11.1.

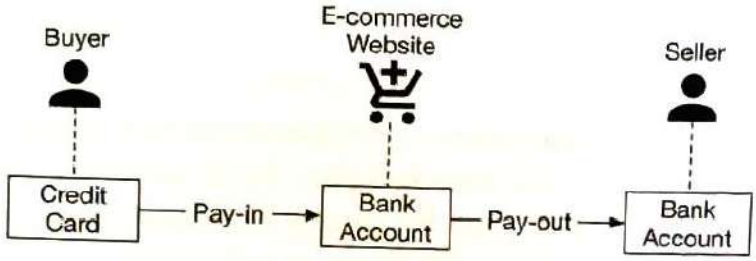


Figure 11.1: Simplified pay-in and pay-out flow

Pay-in flow

The high-level design diagram for the pay-in flow is shown in Figure 11.2. Let's take a look at each component of the system.

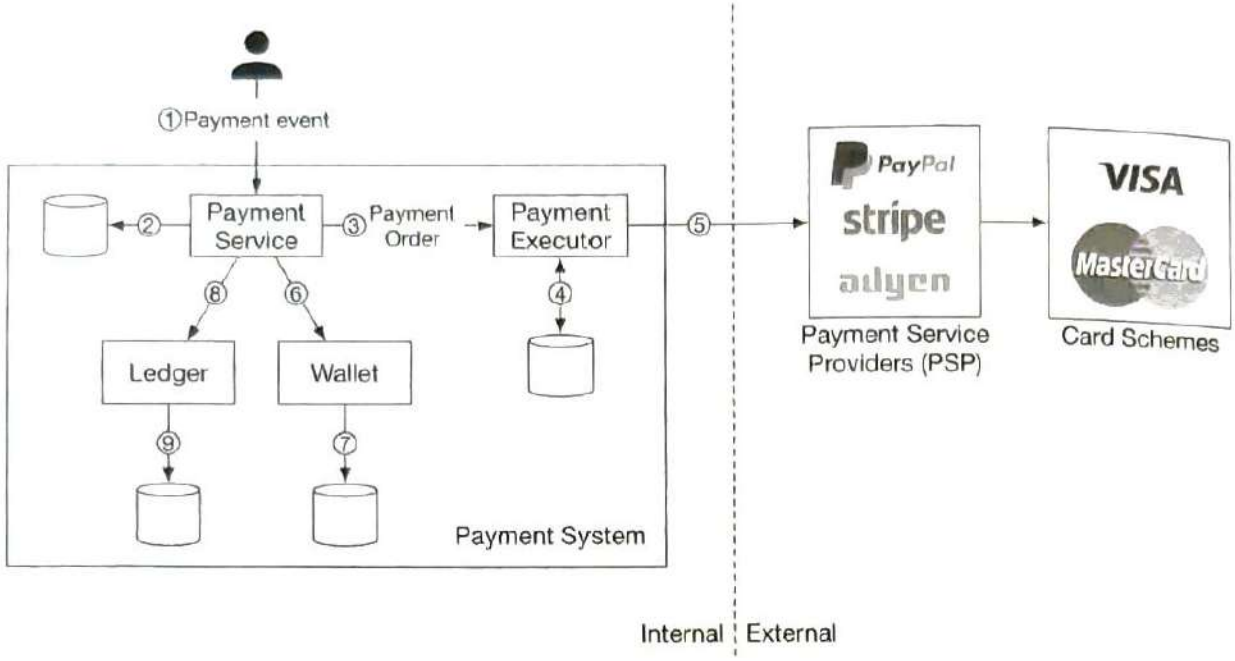


Figure 11.2: Pay-in flow

Payment service

The payment service accepts payment events from users and coordinates the payment process. The first thing it usually does is a risk check, assessing for compliance with regulations such as AML/CFT [2], and for evidence of criminal activity such as money laundering or financing of terrorism. The payment service only processes payments that pass this risk check. Usually, the risk check service uses a third-party provider because it is very complicated and highly specialized.

Payment executor

The payment executor executes a single payment order via a Payment Service Provider (PSP). A payment event may contain several payment orders.

Payment Service Provider (PSP)

A PSP moves money from account A to account B. In this simplified example, the PSP moves the money out of the buyer's credit card account.

Card schemes

Card schemes are the organizations that process credit card operations. Well known card schemes are Visa, MasterCard, Discovery, etc. The card scheme ecosystem is very complex [3].

Ledger

The ledger keeps a financial record of the payment transaction. For example, when a user pays the seller $1, we record it as debit $1 from the user and credit $1 to the seller. The ledger system is very important in post-payment analysis, such as calculating the total revenue of the e-commerce website or forecasting future revenue.

Wallet

The wallet keeps the account balance of the merchant. It may also record how much a given user has paid in total.

As shown in Figure 11.2, a typical pay-in flow works like this:

1. When a user clicks the "place order" button, a payment event is generated and sent to the payment service.
2. The payment service stores the payment event in the database.
3. Sometimes, a single payment event may contain several payment orders. For example, you may select products from multiple sellers in a single checkout process. If the e-commerce website splits the checkout into multiple payment orders, the payment service calls the payment executor for each payment order.
4. The payment executor stores the payment order in the database.
5. The payment executor calls an external PSP to process the credit card payment.
6. After the payment executor has successfully processed the payment, the payment service updates the wallet to record how much money a given seller has.
7. The wallet server stores the updated balance information in the database.
8. After the wallet service has successfully updated the seller's balance information, the payment service calls the ledger to update it.
9. The ledger service appends the new ledger information to the database.

APIs for payment service

We use the RESTful API design convention for the payment service.

`POST /v1/payments`

This endpoint executes a payment event. As mentioned above, a single payment event may contain multiple payment orders. The request parameters are listed below:

Field	Description	Type
`buyer_info`	The information of the buyer	json
`checkout_id`	A globally unique ID for this checkout	string
`credit_card_info`	This could be encrypted credit card information or a payment token. The value is PSP-specific.	json
`payment_orders`	A list of the payment orders	list

Table 11.1: API request parameters (execute a payment event)

The `payment_orders` look like this:

Field	Description	Type
seller_account	Which seller will receive the money	string
amount	The transaction amount for the order	string
currency	The currency for the order	string (ISO 4217 [4])
payment_order_id	A globally unique ID for this payment	string

Table 11.2: payment_orders

Note that the payment_order_id is globally unique. When the payment executor sends a payment request to a third-party PSP, the payment_order_id is used by the PSP as the deduplication ID, also called the idempotency key.

You may have noticed that the data type of the "amount" field is "string," rather than "double". Double is not a good choice because:

1. Different protocols, software, and hardware may support different numeric precisions in serialization and deserialization. This difference might cause unintended rounding errors.
2. The number could be extremely big (for example, Japan's GDP is around 5×10^{14} yen for the calendar year 2020), or extremely small (for example, a satoshi of Bitcoin is 10^{-8}).

It is recommended to keep numbers in string format during transmission and storage. They are only parsed to numbers when used for display or calculation.

GET /v1/payments/{:id}

This endpoint returns the execution status of a single payment order based on payment_order_id.

The payment API mentioned above is similar to the API of some well-known PSPs. If you are interested in a more comprehensive view of payment APIs, check out Stripe's API documentation [5].

The data model for payment service

We need two tables for the payment service: payment event and payment order. When we select a storage solution for a payment system, performance is usually not the most important factor. Instead, we focus on the following:

1. Proven stability. Whether the storage system has been used by other big financial firms for many years (for example more than 5 years) with positive feedback.
2. The richness of supporting tools, such as monitoring and investigation tools.
3. Maturity of the database administrator (DBA) job market. Whether we can recruit experienced DBAs is a very important factor to consider.

Usually, we prefer a traditional relational database with ACID transaction support over NoSQL/NewSQL.

The payment event table contains detailed payment event information. This is what it looks like:

Name	Type
checkout_id	string **PK**
buyer_info	string
seller_info	string
credit_card_info	depends on the card provider
is_payment_done	boolean

Table 11.3: Payment event

The payment order table stores the execution status of each payment order. This is what it looks like:

Name	Type
payment_order_id	String **PK**
buyer_account	string
amount	string
currency	string
checkout_id	string **FK**
payment_order_status	string
ledger_updated	boolean
wallet_updated	boolean

Table 11.4: Payment order

Before we dive into the tables, let's take a look at some background information.

- The `checkout_id` is the foreign key. A single checkout creates a payment event that may contain several payment orders.
- When we call a third-party PSP to deduct money from the buyer's credit card, the money is not directly transferred to the seller. Instead, the money is transferred to the e-commerce website's bank account. This process is called pay-in. When the pay-out condition is satisfied, such as when the products are delivered, the seller initiates a pay-out. Only then is the money transferred from the e-commerce website's bank account to the seller's bank account. Therefore, during the pay-in flow, we only need the buyer's card information, not the seller's bank account information.

In the payment order table (Table 11.4), `payment_order_status` is an enumerated type (enum) that keeps the execution status of the payment order. Execution status includes `NOT_STARTED`, `EXECUTING`, `SUCCESS`, `FAILED`. The update logic is:

1. The initial status of `payment_order_status` is `NOT_STARTED`.

2. When the payment service sends the payment order to the payment executor, the payment_order_status is EXECUTING.
3. The payment service updates the payment_order_status to SUCCESS or FAILED depending on the response of the payment executor.

Once the payment_order_status is SUCCESS, the payment service calls the wallet service to update the seller balance and update the wallet_updated field to TRUE. Here we simplify the design by assuming wallet updates always succeed.

Once it is done, the next step for the payment service is to call the ledger service to update the ledger database by updating the ledger_updated field to TRUE.

When all payment orders under the same checkout_id are processed successfully, the payment service updates the is_payment_done to TRUE in the payment event table. A scheduled job usually runs at a fixed interval to monitor the status of the in-flight payment orders. It sends an alert when a payment order does not finish within a threshold so that engineers can investigate it.

Double-entry ledger system

There is a very important design principle in the ledger system: the double-entry principle (also called double-entry accounting/bookkeeping [6]). Double-entry system is fundamental to any payment system and is key to accurate bookkeeping. It records every payment transaction into two separate ledger accounts with the same amount. One account is debited and the other is credited with the same amount (Table 11.5).

Account	Debit	Credit
buyer	$1	
seller		$1

Table 11.5: Double-entry system

The double-entry system states that the sum of all the transaction entries must be 0. One cent lost means someone else gains a cent. It provides end-to-end traceability and ensures consistency throughout the payment cycle. To find out more about implementing the double-entry system, see Square's engineering blog about immutable double-entry accounting database service [7].

Hosted payment page

Most companies prefer not to store credit card information internally because if they do, they have to deal with complex regulations such as Payment Card Industry Data Security Standard (PCI DSS) [8] in the United States. To avoid handling credit card information, companies use hosted credit card pages provided by PSPs. For websites, it is a widget or an iframe, while for mobile applications, it may be a pre-built page from the payment SDK. Figure 11.3 illustrates an example of the checkout experience with PayPal integration. The key point here is that the PSP provides a hosted payment page that captures the customer card information directly, rather than relying on our payment service.

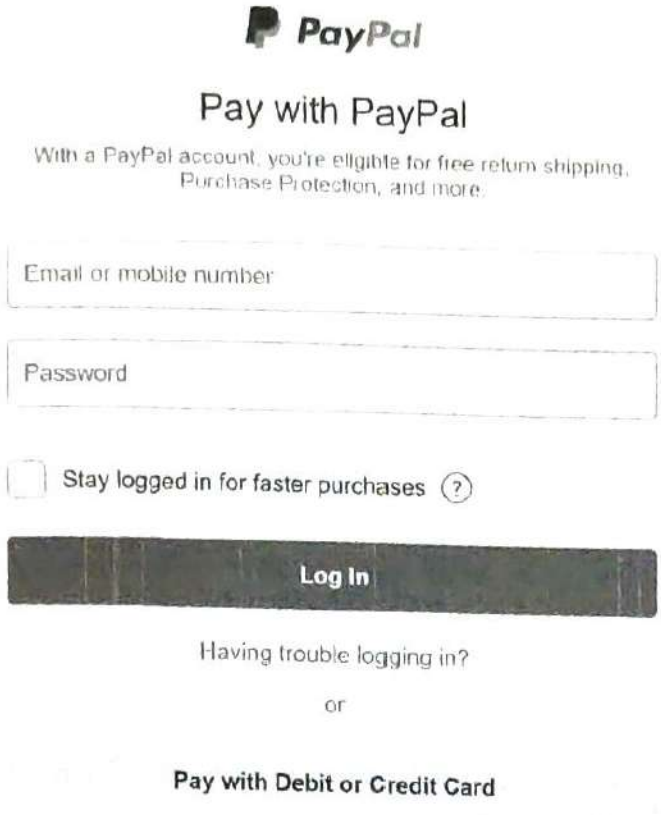


Figure 11.3: Hosted pay with PayPal page

Pay-out flow

The components of the pay-out flow are very similar to the pay-in flow. One difference is that instead of using PSP to move money from the buyer's credit card to the e-commerce website's bank account, the pay-out flow uses a third-party pay-out provider to move money from the e-commerce website's bank account to the seller's bank account.

Usually, the payment system uses third-party account payable providers like Tipalti [9] to handle pay-outs. There are a lot of bookkeeping and regulatory requirements with pay-outs as well.

Step 3 - Design Deep Dive

In this section, we focus on making the system faster, more robust, and secure. In a distributed system, errors and failures are not only inevitable but common. For example, what happens if a customer pressed the "pay" button multiple times? Will they be charged multiple times? How do we handle payment failures caused by poor network connections? In this section, we dive deep into several key topics.

- PSP integration
- Reconciliation
- Handling payment processing delays

- Communication among internal services
- Handling failed payments
- Exact-once delivery
- Consistency
- Security

PSP integration

If the payment system can directly connect to banks or card schemes such as Visa or MasterCard, payment can be made without a PSP. These direct connections are uncommon and highly specialized. They are usually reserved for really large companies that can justify such an investment. For most companies, the payment system integrates with a PSP instead, in one of two ways:

1. If a company can safely store sensitive payment information and chooses to do so, PSP can be integrated using API. The company is responsible for developing the payment web pages, collecting and storing sensitive payment information. PSP is responsible for connecting to banks or card schemes.
2. If a company chooses not to store sensitive payment information due to complex regulations and security concerns, PSP provides a hosted payment page to collect card payment details and securely store them in PSP. This is the approach most companies take.

We use Figure 11.4 to explain how the hosted payment page works in detail.

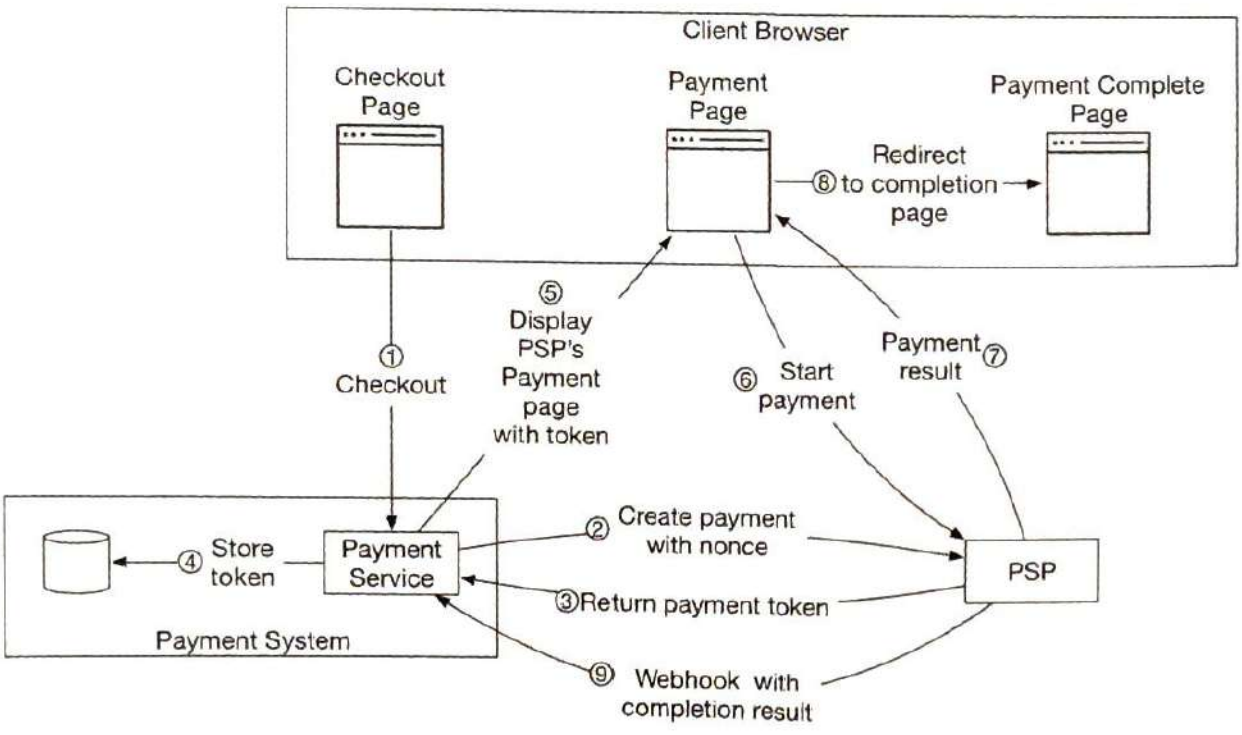


Figure 11.4: Hosted payment flow

We omitted the payment executor, ledger, and wallet in Figure 11.4 for simplicity. The payment service orchestrates the whole payment process.

1. The user clicks the "checkout" button in the client browser. The client calls the payment service with the payment order information.
2. After receiving the payment order information, the payment service sends a payment registration request to the PSP. This registration request contains payment information, such as the amount, currency, expiration date of the payment request, and the redirect URL. Because a payment order should be registered only once, there is a UUID field to ensure the exactly-once registration. This UUID is also called nonce [10]. Usually, this UUID is the ID of the payment order.
3. The PSP returns a token back to the payment service. A token is a UUID on the PSP side that uniquely identifies the payment registration. We can examine the payment registration and the payment execution status later using this token.
4. The payment service stores the token in the database before calling the PSP-hosted payment page.
5. Once the token is persisted, the client displays a PSP-hosted payment page. Mobile applications usually use the PSP's SDK integration for this functionality. Here we use Stripe's web integration as an example (Figure 11.5). Stripe provides a JavaScript library that displays the payment UI, collects sensitive payment information, and calls the PSP directly to complete the payment. Sensitive payment information is collected by Stripe. It never reaches our payment system. The hosted payment page usually needs two pieces of information:
   (a) The token we received in step 4. The PSP's javascript code uses the token to retrieve detailed information about the payment request from the PSP's backend. One important piece of information is how much money to collect.
   (b) Another important piece of information is the redirect URL. This is the web page URL that is called when the payment is complete. When the PSP's JavaScript finishes the payment, it redirects the browser to the redirect URL. Usually, the redirect URL is an e-commerce web page that shows the status of the checkout. Note that the redirect URL is different from the webhook [11] URL in step 9.

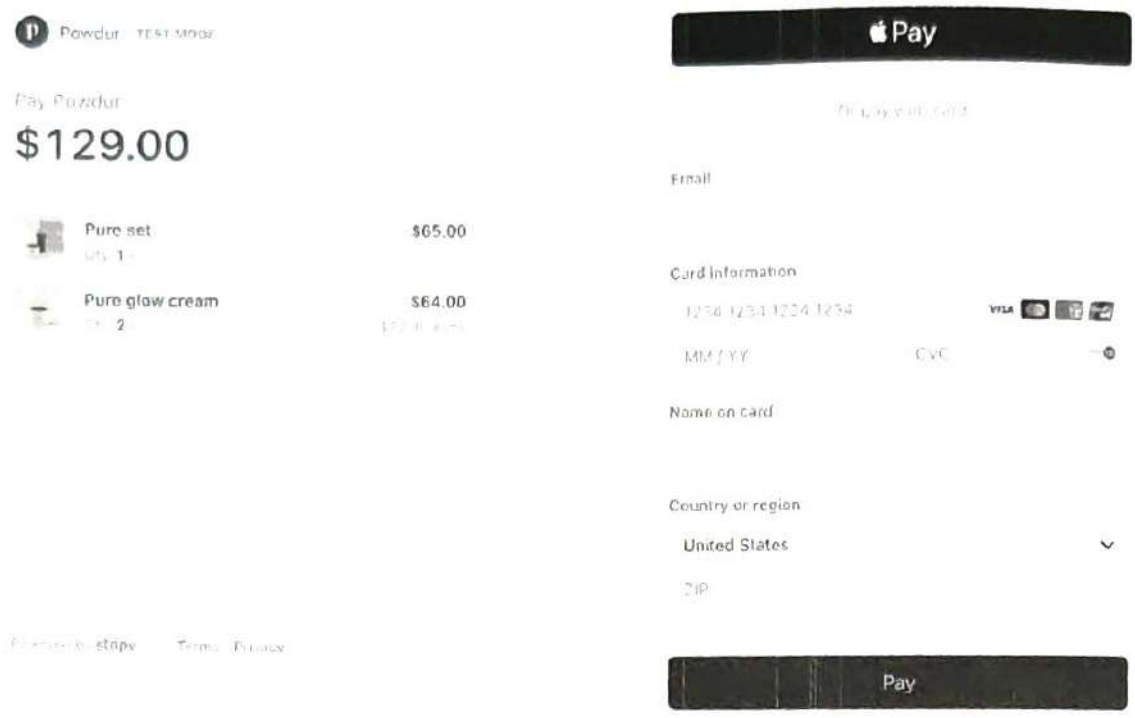


Figure 11.5: Hosted payment page by Stripe

6. The user fills in the payment details on the PSP's web page, such as the credit card number, holder's name, expiration date, etc, then clicks the pay button. The PSP starts the payment processing.
7. The PSP returns the payment status.
8. The web page is now redirected to the redirect URL. The payment status that is received in step 7 is typically appended to the URL. For example, the full redirect URL could be [12]: `https://your-company.com/?tokenID=JIOUIQ123NSF&payResult=X324FSa`
9. Asynchronously, the PSP calls the payment service with the payment status via a webhook. The webhook is an URL on the payment system side that was registered with the PSP during the initial setup with the PSP. When the payment system receives payment events through the webhook, it extracts the payment status and updates the `payment_order_status` field in the Payment Order database table.

So far, we explained the happy path of the hosted payment page. In reality, the network connection could be unreliable and all 9 steps above could fail. Is there any systematic way to handle failure cases? The answer is reconciliation.

Reconciliation

When system components communicate asynchronously, there is no guarantee that a message will be delivered, or a response will be returned. This is very common in the payment business, which often uses asynchronous communication to increase system performance. External systems, such as PSPs or banks, prefer asynchronous communi-

cation as well. So how can we ensure correctness in this case?

The answer is reconciliation. This is a practice that periodically compares the states among related services in order to verify that they are in agreement. It is usually the last line of defense in the payment system.

Every night the PSP or banks send a settlement file to their clients. The settlement file contains the balance of the bank account, together with all the transactions that took place on this bank account during the day. The reconciliation system parses the settlement file and compares the details with the ledger system. Figure 11.6 below shows where the reconciliation process fits in the system.

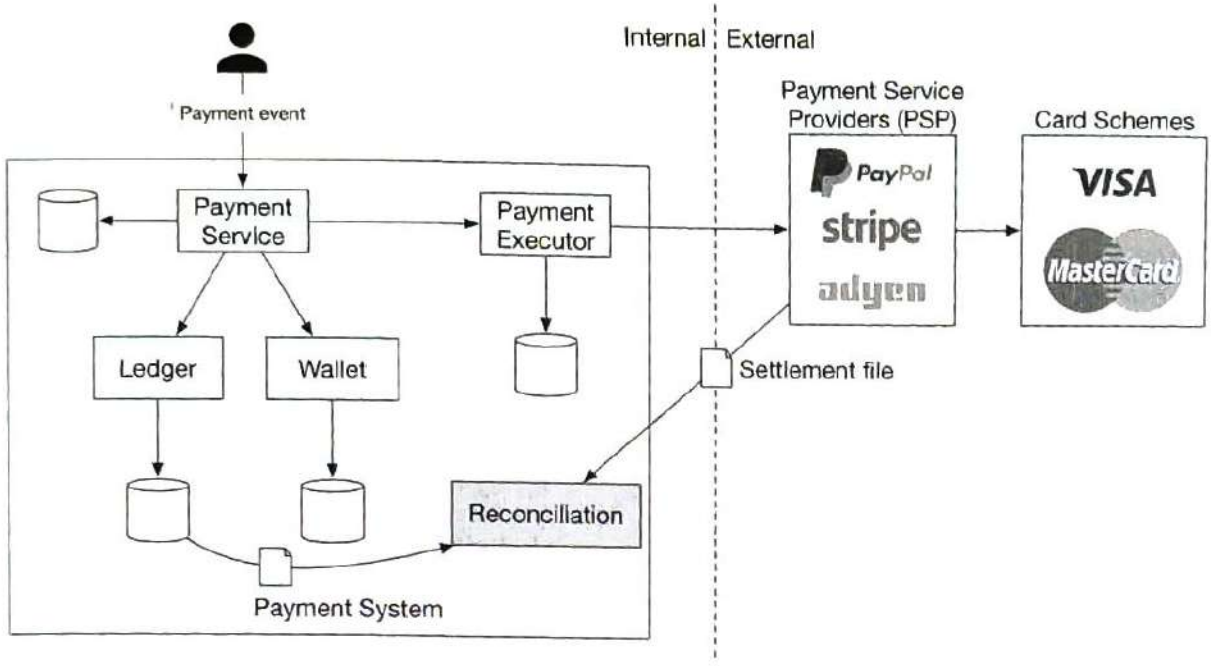


Figure 11.6: Reconciliation

Reconciliation is also used to verify that the payment system is internally consistent. For example, the states in the ledger and wallet might diverge and we could use the reconciliation system to detect any discrepancy.

To fix mismatches found during reconciliation, we usually rely on the finance team to perform manual adjustments. The mismatches and adjustments are usually classified into three categories:

1. The mismatch is classifiable and the adjustment can be automated. In this case, we know the cause of the mismatch, how to fix it, and it is cost-effective to write a program to automate the adjustment. Engineers can automate both the mismatch classification and adjustment.
2. The mismatch is classifiable, but we are unable to automate the adjustment. In this case, we know the cause of the mismatch and how to fix it, but the cost of writing an auto adjustment program is too high. The mismatch is put into a job queue and the finance team fixes the mismatch manually.

3. The mismatch is unclassifiable. In this case, we do not know how the mismatch happens. The mismatch is put into a special job queue. The finance team investigates it manually.

Handling payment processing delays

As discussed previously, an end-to-end payment request flows through many components and involves both internal and external parties. While in most cases a payment request would complete in seconds, there are situations where a payment request would stall and sometimes take hours or days before it is completed or rejected. Here are some examples where a payment request could take longer than usual:

- The PSP deems a payment request high risk and requires a human to review it.
- A credit card requires extra protection like 3D Secure Authentication [13] which requires extra details from a card holder to verify a purchase.

The payment service must be able to handle these payment requests that take a long time to process. If the buy page is hosted by an external PSP, which is quite common these days, the PSP would handle these long-running payment requests in the following ways:

- The PSP would return a pending status to our client. Our client would display that to the user. Our client would also provide a page for the customer to check the current payment status.
- The PSP tracks the pending payment on our behalf, and notifies the payment service of any status update via the webhook the payment service registered with the PSP.

When the payment request is finally completed, the PSP calls the registered webhook mentioned above. The payment service updates its internal system and completes the shipment to the customer.

Alternatively, instead of updating the payment service via a webhook, some PSP would put the burden on the payment service to poll the PSP for status updates on any pending payment requests.

Communication among internal services

There are two types of communication patterns that internal services use to communicate: synchronous vs asynchronous. Both are explained below.

Synchronous communication

Synchronous communication like HTTP works well for small-scale systems, but its shortcomings become obvious as the scale increases. It creates a long request and response cycle that depends on many services. The drawbacks of this approach are:

- Low performance. If any one of the services in the chain doesn't perform well, the whole system is impacted.

- Poor failure isolation. If PSPs or any other services fail, the client will no longer receive a response.
- Tight coupling. The request sender needs to know the recipient.
- Hard to scale. Without using a queue to act as a buffer, it's not easy to scale the system to support a sudden increase in traffic.

Asynchronous communication

Asynchronous communication can be divided into two categories:

- Single receiver: each request (message) is processed by one receiver or service. It's usually implemented via a shared message queue. The message queue can have multiple subscribers, but once a message is processed, it gets removed from the queue. Let's take a look at a concrete example. In Figure 11.7, service A and service B both subscribe to a shared message queue. When m1 and m2 are consumed by service A and service B respectively, both messages are removed from the queue as shown in Figure 11.8.

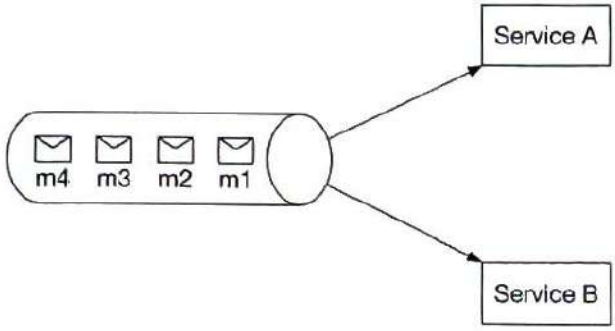


Figure 11.7: Message queue

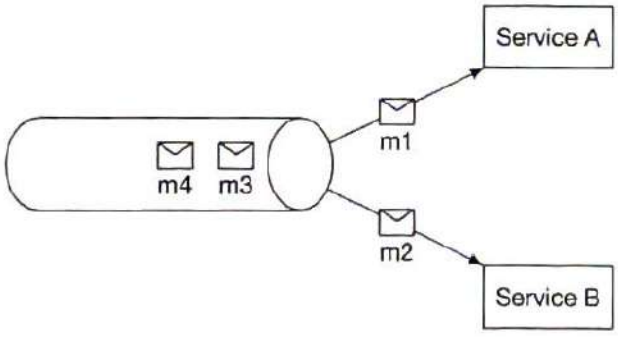


Figure 11.8: Single receiver for each message

- Multiple receivers: each request (message) is processed by multiple receivers or services. Kafka works well here. When consumers receive messages, they are not removed from Kafka. The same message can be processed by different services. This model maps well to the payment system, as the same request might trigger multiple side effects such as sending push notifications, updating financial reporting, ana-

lytics, etc. An example is illustrated in Figure 11.9. Payment events are published to Kafka and consumed by different services such as the payment system, analytics service, and billing service.

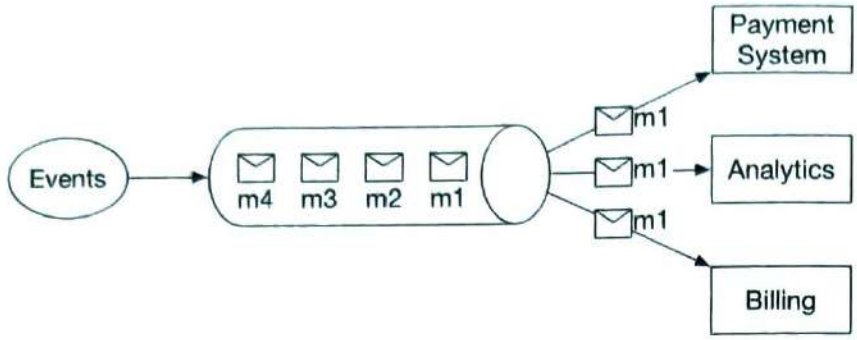


Figure 11.9: Multiple receivers for the same message

Generally speaking, synchronous communication is simpler in design, but it doesn't allow services to be autonomous. As the dependency graph grows, the overall performance suffers. Asynchronous communication trades design simplicity and consistency for scalability and failure resilience. For a large-scale payment system with complex business logic and a large number of third-party dependencies, asynchronous communication is a better choice.

Handling failed payments

Every payment system has to handle failed transactions. Reliability and fault tolerance are key requirements. We review some of the techniques for tackling those challenges.

Tracking payment state

Having a definitive payment state at any stage of the payment cycle is crucial. Whenever a failure happens, we can determine the current state of a payment transaction and decide whether a retry or refund is needed. The payment state can be persisted in an append-only database table.

Retry queue and dead letter queue

To gracefully handle failures, we utilize the retry queue and dead letter queue, as shown in Figure 11.10.

- Retry queue: retryable errors such as transient errors are routed to a retry queue.
- Dead letter queue [14]: if a message fails repeatedly, it eventually lands in the dead letter queue. A dead letter queue is useful for debugging and isolating problematic messages for inspection to determine why they were not processed successfully.

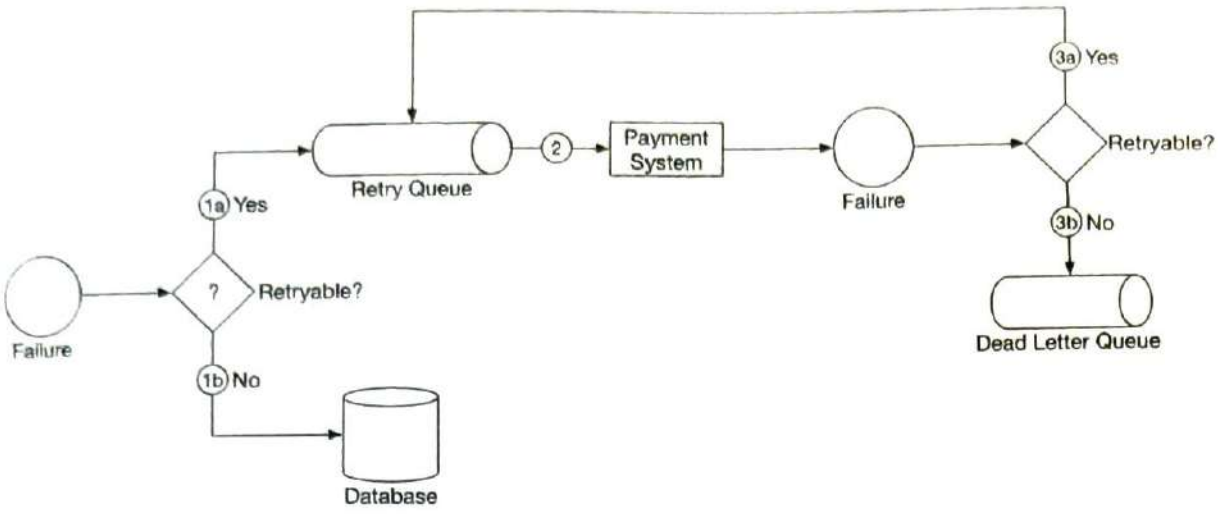


Figure 11.10: Handle failed payments

1. Check whether the failure is retryable.
   (a) Retryable failures are routed to a retry queue.
   (b) For non-retryable failures such as invalid input, errors are stored in a database.
2. The payment system consumes events from the retry queue and retries failed payment transactions.
3. If the payment transaction fails again:
   (a) If the retry count doesn't exceed the threshold, the event is routed to the retry queue.
   (b) If the retry count exceeds the threshold, the event is put in the dead letter queue. Those failed events might need to be investigated.

If you are interested in a real-world example of using those queues, take a look at Uber's payment system that utilizes Kafka to meet the reliability and fault-tolerance requirements [15].

Exactly-once delivery

One of the most serious problems a payment system can have is to double charge a customer. It is important to guarantee in our design that the payment system executes a payment order exactly-once [16].

At first glance, exactly-once delivery seems very hard to tackle, but if we divide the problem into two parts, it is much easier to solve. Mathematically, an operation is executed exactly-once if:

1. It is executed at-least-once.
2. At the same time, it is executed at-most-once.

We will explain how to implement at-least-once using retry, and at-most-once using idempotency check.

Retry

Occasionally, we need to retry a payment transaction due to network errors or timeout. Retry provides the at-least-once guarantee. For example, as shown in Figure 11.11, where the client tries to make a $10 payment, but the payment request keeps failing due to a poor network connection. In this example, the network eventually recovered and the request succeeded at the fourth attempt.

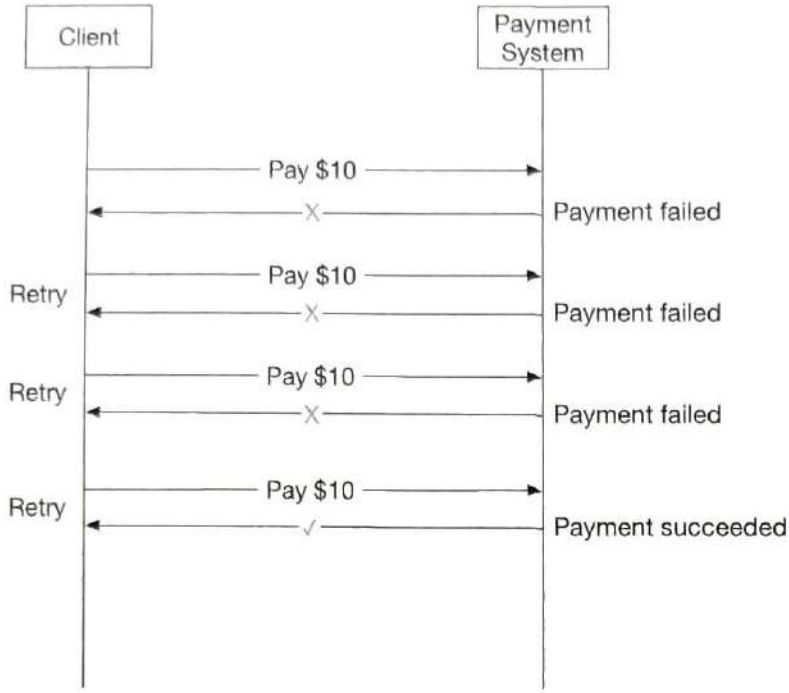


Figure 11.11: Retry

Deciding the appropriate time intervals between retries is important. Here are a few common retry strategies.

- Immediate retry: client immediately resends a request.
- Fixed intervals: wait a fixed amount of time between the time of the failed payment and a new retry attempt.
- Incremental intervals: client waits for a short time for the first retry, and then incrementally increases the time for subsequent retries.
- Exponential backoff [17]: double the waiting time between retries after each failed retry. For example, when a request fails for the first time, we retry after 1 second; if it fails a second time, we wait 2 seconds before the next retry; if it fails a third time, we wait 4 seconds before another retry.
- Cancel: the client can cancel the request. This is a common practice when the failure is permanent or repeated requests are unlikely to be successful.

Determining the appropriate retry strategy is difficult. There is no "one size fits all" solution. As a general guideline, use exponential backoff if the network issue is unlikely to be resolved in a short amount of time. Overly aggressive retry strategies waste computing

resources and can cause service overload. A good practice is to provide an error code with a `Retry-After` header.

A potential problem of retrying is double payments. Let us take a look at two scenarios.

Scenario 1: The payment system integrates with PSP using a hosted payment page, and the client clicks the pay button twice.

Scenario 2: The payment is successfully processed by the PSP, but the response fails to reach our payment system due to network errors. The user clicks the "pay" button again or the client retries the payment.

In order to avoid double payment, the payment has to be executed at-most-once. This at-most-once guarantee is also called idempotency.

Idempotency

Idempotency is key to ensuring the at-most-once guarantee. According to Wikipedia, "idempotence is the property of certain operations in mathematics and computer science whereby they can be applied multiple times without changing the result beyond the initial application" [18]. From an API standpoint, idempotency means clients can make the same call repeatedly and produce the same result.

For communication between clients (web and mobile applications) and servers, an idempotency key is usually a unique value that is generated by the client and expires after a certain period of time. A UUID is commonly used as an idempotency key and it is recommended by many tech companies such as Stripe [19] and PayPal [20]. To perform an idempotent payment request, an idempotency key is added to the HTTP header: `<idempotency-key: key_value>`.

Now that we understand the basics of idempotency, let's take a look at how it helps to solve the double payment issues mentioned above.

Scenario 1: what if a customer clicks the "pay" button quickly twice?

In Figure 11.12, when a user clicks "pay," an idempotency key is sent to the payment system as part of the HTTP request. In an e-commerce website, the idempotency key is usually the ID of the shopping cart right before the checkout.

For the second request, it's treated as a retry because the payment system has already seen the idempotency key. When we include a previously specified idempotency key in the request header, the payment system returns the latest status of the previous request.

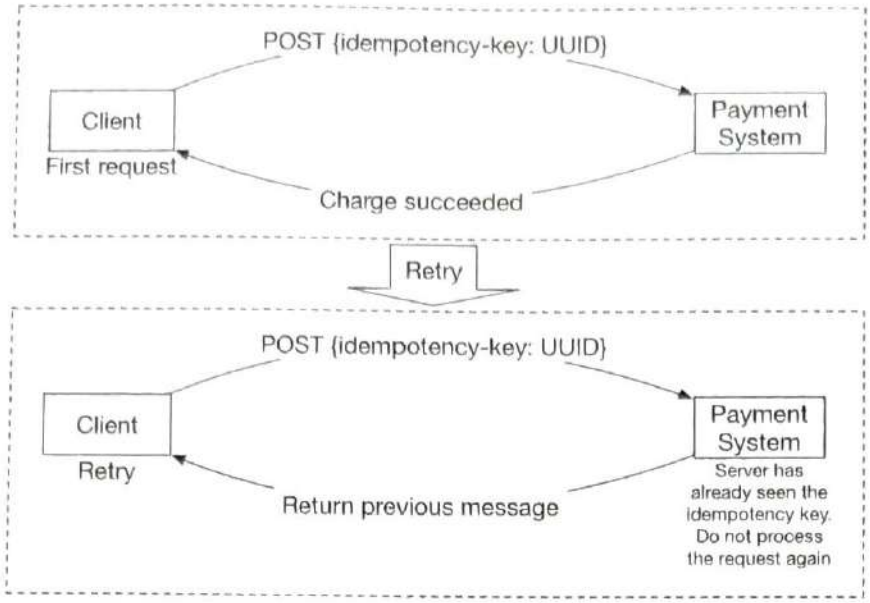


Figure 11.12: Idempotency

If multiple concurrent requests are detected with the same idempotency key, only one request is processed and the others receive the `429 Too Many Requests` status code.

To support idempotency, we can use the database's unique key constraint. For example, the primary key of the database table is served as the idempotency key. Here is how it works:

1. When the payment system receives a payment, it tries to insert a row into the database table.
2. A successful insertion means we have not seen this payment request before.
3. If the insertion fails because the same primary key already exists, it means we have seen this payment request before. The second request will not be processed.

Scenario 2: The payment is successfully processed by the PSP, but the response fails to reach our payment system due to network errors. Then the user clicks the "pay" button again.

As shown in Figure 11.4 (step 2 and step 3), the payment service sends the PSP a nonce and the PSP returns a corresponding token. The nonce uniquely represents the payment order, and the token uniquely maps to the nonce. Therefore, the token uniquely maps to the payment order.

When the user clicks the "pay" button again, the payment order is the same, so the token sent to the PSP is the same. Because the token is used as the idempotency key on the PSP side, it is able to identify the double payment and return the status of the previous execution.

Consistency

Several stateful services are called in a payment execution:

1. The payment service keeps payment-related data such as nonce, token, payment order, execution status, etc.
2. The ledger keeps all accounting data.
3. The wallet keeps the account balance of the merchant.
4. The PSP keeps the payment execution status.
5. Data might be replicated among different database replicas to increase reliability.

In a distributed environment, the communication between any two services can fail, causing data inconsistency. Let's take a look at some techniques to resolve data inconsistency in a payment system.

To maintain data consistency between internal services, ensuring exactly-once processing is very important.

To maintain data consistency between the internal service and external service (PSP), we usually rely on idempotency and reconciliation. If the external service supports idempotency, we should use the same idempotency key for payment retry operations. Even if an external service supports idempotent APIs, reconciliation is still needed because we shouldn't assume the external system is always right.

If data is replicated, replication lag could cause inconsistent data between the primary database and the replicas. There are generally two options to solve this:

1. Serve both reads and writes from the primary database only. This approach is easy to set up, but the obvious drawback is scalability. Replicas are used to ensure data reliability, but they don't serve any traffic, which wastes resources.
2. Ensure all replicas are always in-sync. We could use consensus algorithms such as Paxos [21] and Raft [22], or use consensus-based distributed databases such as YugabyteDB [23] or CockroachDB [24].

Payment security

Payment security is very important. In the final part of this system design, we briefly cover a few techniques for combating cyberattacks and card thefts.

Problem	Solution
Request/response eavesdropping	Use HTTPS
Data tampering	Enforce encryption and integrity monitoring
Man-in-the-middle attack	Use SSL with certificate pinning
Data loss	Database replication across multiple regions and take snapshots of data
Distributed denial-of-service attack (DDoS)	Rate limiting and firewall [25]
Card theft	Tokenization. Instead of using real card numbers, tokens are stored and used for payment
PCI compliance	PCI DSS is an information security standard for organizations that handle branded credit cards
Fraud	Address verification, card verification value (CVV), user behavior analysis, etc. [26] [27]

Table 11.6: Payment security

Step 4 - Wrap Up

In this chapter, we investigated the pay-in flow and pay-out flow. We went into great depth about retry, idempotency, and consistency. Payment error handling and security are also covered at the end of the chapter.

A payment system is extremely complex. Even though we have covered many topics, there are still more worth mentioning. The following is a representative but not an exhaustive list of relevant topics.

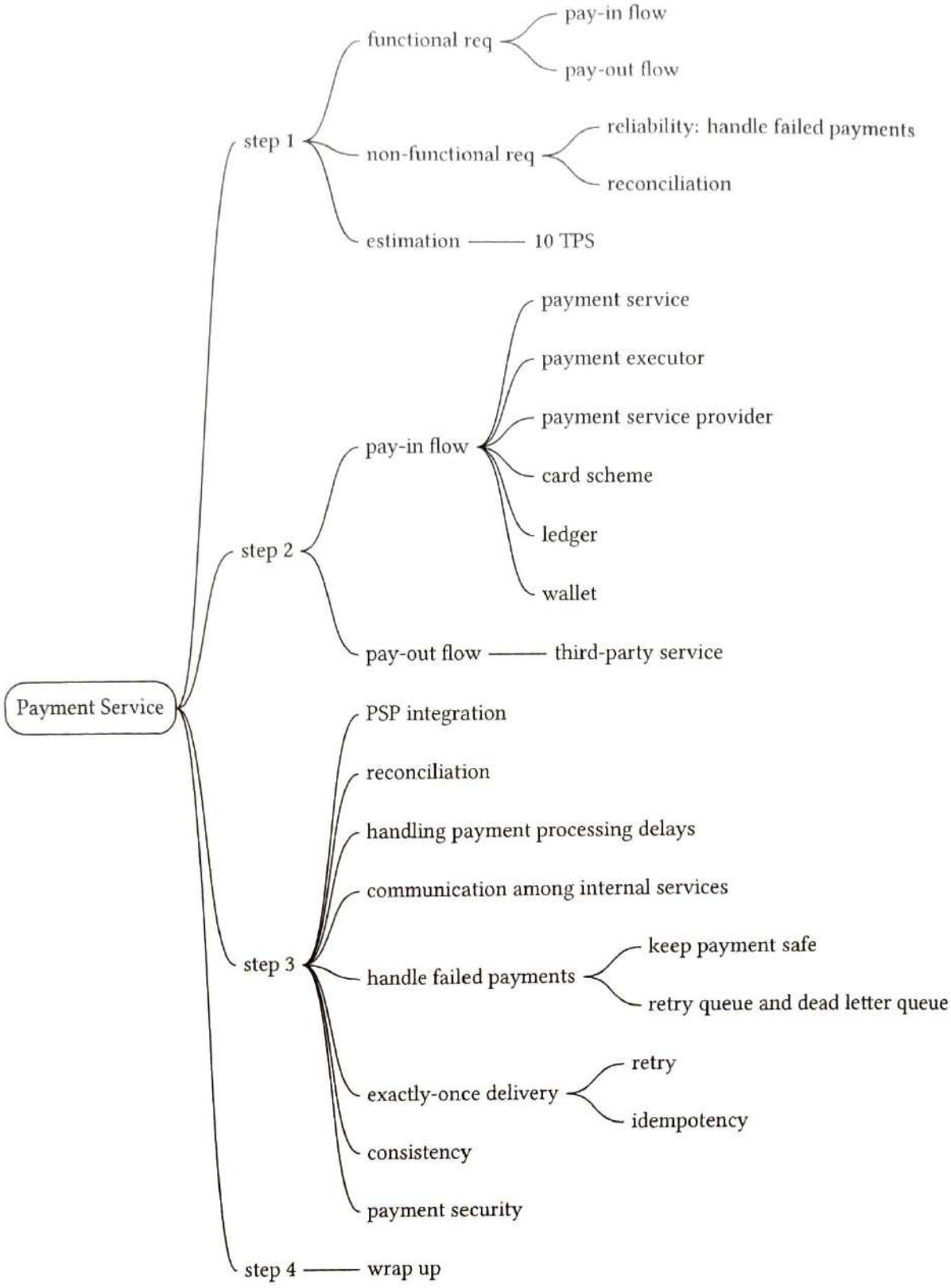
- Monitoring. Monitoring key metrics is a critical part of any modern application. With extensive monitoring, we can answer questions like "What is the average acceptance rate for a specific payment method?", "What is the CPU usage of our servers?", etc. We can create and display those metrics on a dashboard.
- Alerting. When something abnormal occurs, it is important to alert on-call developers so they respond promptly.
- Debugging tools. "Why does a payment fail?" is a common question. To make debugging easier for engineers and customer support, it is important to develop tools that allow staff to review the transaction status, processing server history, PSP records, etc. of a payment transaction.
- Currency exchange. Currency exchange is an important consideration when designing a payment system for an international user base.
- Geography. Different regions might have completely different sets of payment meth-

ods.

- Cash payment. Cash payment is very common in India, Brazil, and some other countries. Uber [28] and Airbnb [29] wrote detailed engineering blogs about how they handled cash-based payment.
- Google/Apple pay integration. Please read [30] if interested.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Chapter Summary



Reference Material

[1] Payment system. https://en.wikipedia.org/wiki/Payment_system.

[2] AML/CFT. https://en.wikipedia.org/wiki/Money_laundering.

[3] Card scheme. https://en.wikipedia.org/wiki/Card_scheme.

[4] ISO 4217. https://en.wikipedia.org/wiki/ISO_4217.

[5] Stripe API Reference. https://stripe.com/docs/api.

[6] Double-entry bookkeeping. https://en.wikipedia.org/wiki/Double-entry_bookkeeping.

[7] Books, an immutable double-entry accounting database service. https://developer.squareup.com/blog/books-an-immutable-double-entry-accounting-database-service/.

[8] Payment Card Industry Data Security Standard. https://en.wikipedia.org/wiki/Payment_Card_Industry_Data_Security_Standard.

[9] Tipalti. https://tipalti.com/.

[10] Nonce. https://en.wikipedia.org/wiki/Cryptographic_nonce.

[11] Webhooks. https://stripe.com/docs/webhooks.

[12] Customize your success page. https://stripe.com/docs/payments/checkout/custom-success-page.

[13] 3D Secure. https://en.wikipedia.org/wiki/3-D_Secure.

[14] Kafka Connect Deep Dive – Error Handling and Dead Letter Queues. https://www.confluent.io/blog/kafka-connect-deep-dive-error-handling-dead-letter-queues/.

[15] Reliable Processing in a Streaming Payment System. https://www.youtube.com/watch?v=5TD8m7w1xE0&list=PLLEUtp5eGr7Dz3fWGUpiSiG3d_WgJe-KJ.

[16] Chain Services with Exactly-Once Guarantees. https://www.confluent.io/blog/chain-services-exactly-guarantees/.

[17] Exponential backoff. https://en.wikipedia.org/wiki/Exponential_backoff.

[18] Idempotence. https://en.wikipedia.org/wiki/Idempotence.

[19] Stripe idempotent requests. https://stripe.com/docs/api/idempotent_requests.

[20] Idempotency. https://developer.paypal.com/docs/platforms/develop/idempotency/.

[21] Paxos. https://en.wikipedia.org/wiki/Paxos_(computer_science).

[22] Raft. https://raft.github.io/.

[23] YogabyteDB. https://www.yugabyte.com/.

[24] Cockroachdb. https://www.cockroachlabs.com/.

[25] What is DDoS attack. https://www.cloudflare.com/learning/ddos/what-is-a-ddos-attack/.

[26] How Payment Gateways Can Detect and Prevent Online Fraud. https://www.chargebee.com/blog/optimize-online-billing-stop-online-fraud/.

[27] Advanced Technologies for Detecting and Preventing Fraud at Uber. https://eng.uber.com/advanced-technologies-detecting-preventing-fraud-uber/.

[28] Re-Architecting Cash and Digital Wallet Payments for India with Uber Engineering. https://eng.uber.com/india-payments/.

[29] Scaling Airbnb's Payment Platform. https://medium.com/airbnb-engineering/scaling-airbnbs-payment-platform-43ebfc99b324.

[30] Payments Integration at Uber: A Case Study – Gergely Orosz. https://www.youtube.com/watch?v=yooCE5B0SRA.

12 Digital Wallet

Payment platforms usually provide a digital wallet service to clients, so they can store money in the wallet and spend it later. For example, you can add money to your digital wallet from your bank card and when you buy products online, you are given the option to pay using the money in your wallet. Figure 12.1 shows this process.

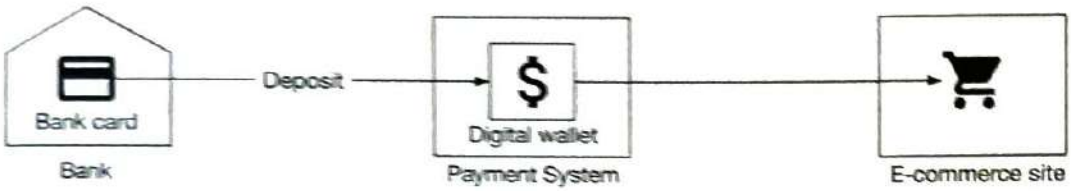


Figure 12.1: Digital wallet

Spending money is not the only feature that the digital wallet provides. For a payment platform like PayPal, we can directly transfer money to somebody else's wallet on the same payment platform. Compared with the bank-to-bank transfer, direct transfer between digital wallets is faster, and most importantly, it usually does not charge an extra fee. Figure 12.2 shows a cross-wallet balance transfer operation.

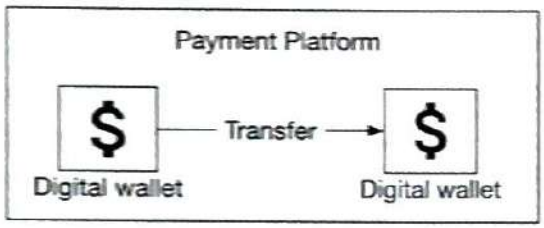


Figure 12.2: Cross-wallet balance transfer

Suppose we are asked to design the backend of a digital wallet application that supports the cross-wallet balance transfer operation. At the beginning of the interview, we will ask clarification questions to nail down the requirements.

Step 1 - Understand the Problem and Establish Design Scope

Candidate: Should we only focus on balance transfer operations between two digital wallets? Do we need to worry about other features?
Interviewer: Let's focus on balance transfer operations only.

Candidate: How many transactions per second (TPS) does the system need to support?
Interviewer: Let's assume 1,000,000 TPS.

Candidate: A digital wallet has strict requirements for correctness. Can we assume transactional guarantees [1] are sufficient?
Interviewer: That sounds good.

Candidate: Do we need to prove correctness?
Interviewer: This is a good question. Correctness is usually only verifiable after a transaction is complete. One way to verify is to compare our internal records with statements from banks. The limitation of reconciliation is that it only shows discrepancies and cannot tell how a difference was generated. Therefore, we would like to design a system with reproducibility, meaning we could always reconstruct historical balance by replaying the data from the very beginning.

Candidate: Can we assume the availability requirement is 99.99%
Interviewer: Sounds good.

Candidate: Do we need to take foreign exchange into consideration?
Interviewer: No, it's out of scope.

In summary, our digital wallet needs to support the following:

- Support balance transfer operation between two digital wallets.
- Support 1,000,000 TPS.
- Reliability is at least 99.99%.
- Support transactions.
- Support reproducibility.

Back-of-the-envelope estimation

When we talk about TPS, we imply a transactional database will be used. Today, a relational database running on a typical data center node can support a few thousand transactions per second. For example, reference [2] contains the performance benchmark of some of the popular transactional database servers. Let's assume a database node can support 1,000 TPS. In order to reach 1 million TPS, we need 1,000 database nodes.

However, this calculation is slightly inaccurate. Each transfer command requires two operations: deducting money from one account and depositing money to the other account. To support 1 million transfers per second, the system actually needs to handle up to 2 million TPS, which means we need 2,000 nodes.

Table 12.1 shows the total number of nodes required when the "per-node TPS" (the TPS a single node can handle) changes. Assuming hardware remains the same, the more

transactions a single node can handle per second, the lower the total number of nodes required, indicating lower hardware cost. So one of our design goals is to increase the number of transactions a single node can handle.

Per-node TPS	Node Number
100	20,000
1,000	2,000
10,000	200

Table 12.1: Mapping between pre-node TPS and node number

Step 2 - Propose High-level Design and Get Buy-in

In this section, we will discuss the following:

- API design
- Three high-level designs
  1. Simple in-memory solution
  2. Database-based distributed transaction solution
  3. Event sourcing solution with reproducibility

API design

We will use the RESTful API convention. For this interview, we only need to support one API:

API	Detail
`POST /v1/wallet/balance_transfer`	Transfer balance from one wallet to another

Request parameters are:

Field	Description	Type
`from_account`	The debit account	string
`to_account`	The credit account	string
`amount`	The amount of money	string
`currency`	The currency type	string (ISO 4217 [3])
`transaction_id`	ID used for deduplication	uuid

Sample response body:

```
{
  "Status": "success"
  "Transaction_id": "01589980-2664-11ec-9621-0242ac130002"
}
```

One thing worth mentioning is that the data type of the "amount" field is "string,"

rather than "double". We explained the reasoning in Chapter 11 Payment System on page 320.

In practice, many people still choose float or double representation of numbers because it is supported by almost every programming language and database. It is a proper choice as long as we understand the potential risk of losing precision.

In-memory sharding solution

The wallet application maintains an account balance for every user account. A good data structure to represent this <user,balance> relationship is a map, which is also called a hash table (map) or key-value store.

For in-memory stores, one popular choice is Redis. One Redis node is not enough to handle 1 million TPS. We need to set up a cluster of Redis nodes and evenly distribute user accounts among them. This process is called partitioning or sharding.

To distribute the key-value data among n partitions, we could calculate the hash value of the key and divide it by n. The remainder is the destination of the partition. The pseudocode below shows the sharding process:

```
String accountID = "A";
Int partitionNumber = 7;
Int myPartition = accountID.hashCode() % partitionNumber;
```

The number of partitions and addresses of all Redis nodes can be stored in a centralized place. We could use ZooKeeper [4] as a highly-available configuration storage solution.

The final component of this solution is a service that handles the transfer commands. We call it the wallet service and it has several key responsibilities.

1. Receives the transfer command
2. Validates the transfer command
3. If the command is valid, it updates the account balances for the two users involved in the transfer. In a cluster, the account balances are likely to be in different Redis nodes

The wallet service is stateless. It is easy to scale horizontally. Figure 12.3 shows the in-memory solution.

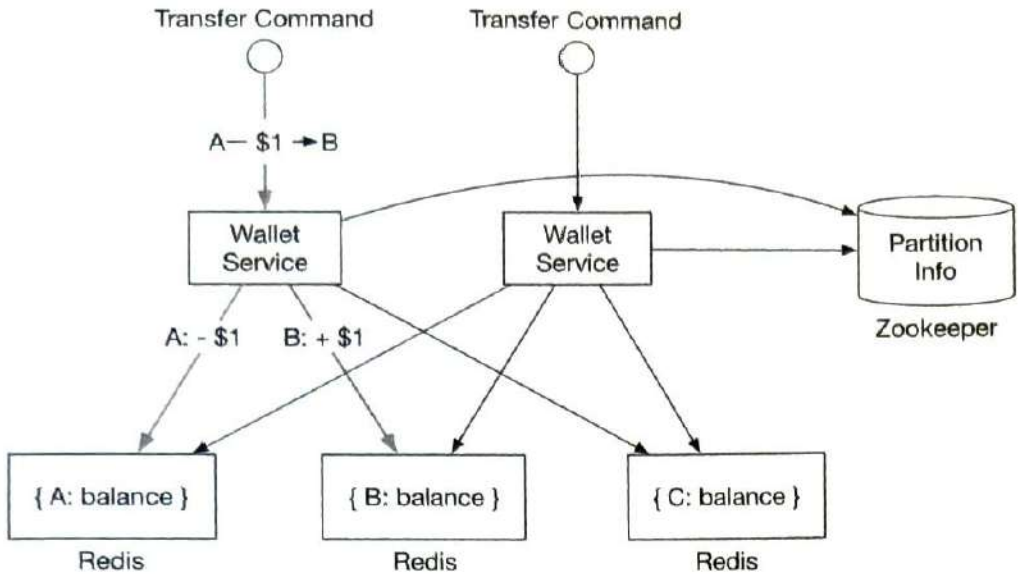


Figure 12.3: In-memory solution

In this example, we have 3 Redis nodes. There are three clients, A, B, and C. Their account balances are evenly spread across these three Redis nodes. There are two wallet service nodes in this example that handle the balance transfer requests. When one of the wallet service nodes receives the transfer command which is to move $1 from client A to client B, it issues two commands to two Redis nodes. For the Redis node that contains client A's account, the wallet service deducts $1 from the account. For client B, the wallet service adds $1 to the account.

Candidate: In this design, account balances are spread across multiple Redis nodes. ZooKeeper is used to maintain the sharding information. The stateless wallet service uses the sharding information to locate the Redis nodes for the clients and updates the account balances accordingly.

Interviewer: This design works, but it does not meet our correctness requirement. The wallet service updates two Redis nodes for each transfer. There is no guarantee that both updates would succeed. If, for example, the wallet service node crashes after the first update has gone through but before the second update is done, it would result in an incomplete transfer. The two updates need to be in a single atomic transaction.

Distributed transactions

Database sharding

How do we make the updates to two different storage nodes atomic? The first step is to replace each Redis node with a transactional relational database node. Figure 12.4 shows the architecture. This time, clients A, B, and C are partitioned into 3 relational databases, rather than in 3 Redis nodes.

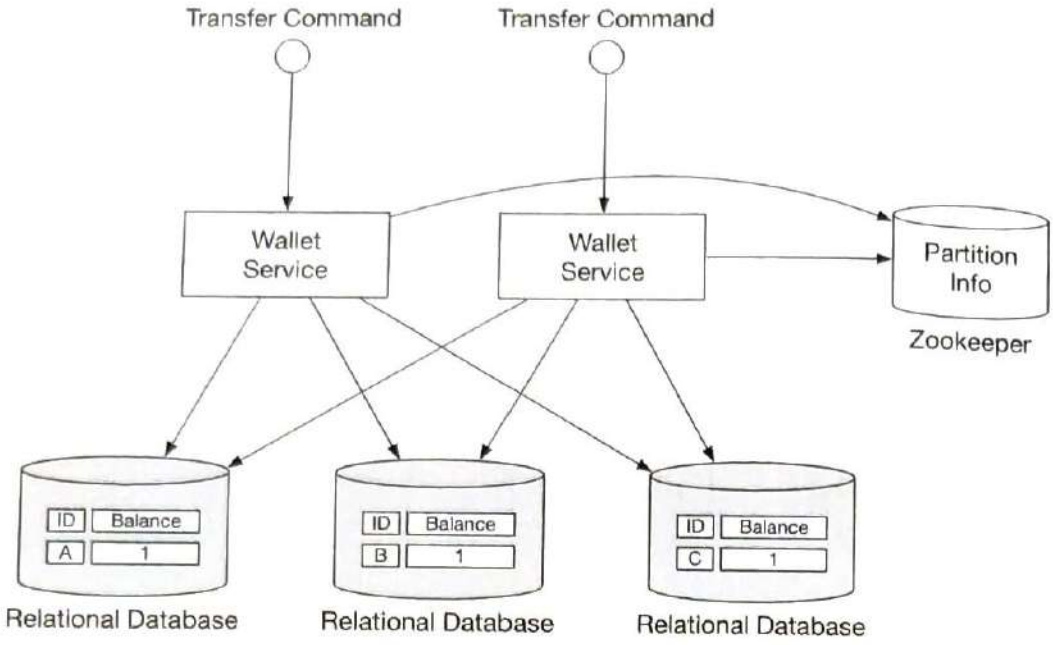


Figure 12.4: Relational database

Using transactional databases only solves part of the problem. As mentioned in the last section, it is very likely that one transfer command will need to update two accounts in two different databases. There is no guarantee that two update operations will be handled at exactly the same time. If the wallet service restarted right after it updated the first account balance, how can we make sure the second account will be updated as well?

Distributed transaction: Two-phase commit

In a distributed system, a transaction may involve multiple processes on multiple nodes. To make a transaction atomic, the distributed transaction might be the answer. There are two ways to implement a distributed transaction: a low-level solution and a high-level solution. We will examine each of them.

The low-level solution relies on the database itself. The most commonly used algorithm is called two-phase commit (2PC). As the name implies, it has two phases, as in Figure 12.5.

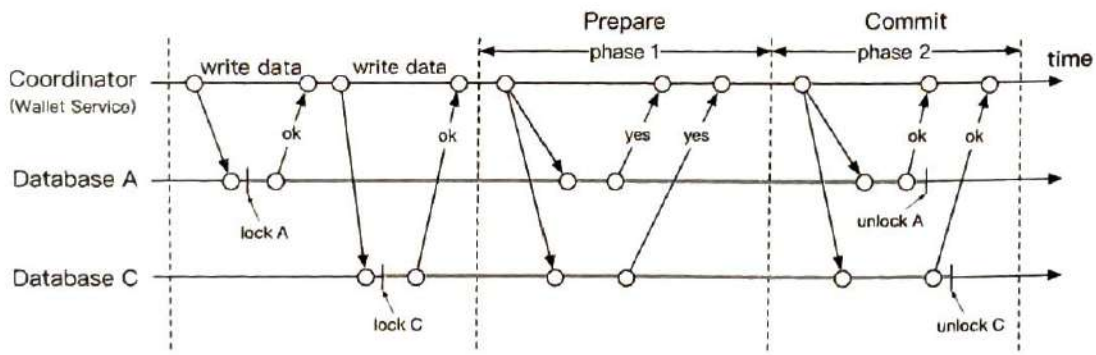


Figure 12.5: Two-phase commit (source [5])

1. The coordinator, which in our case is the wallet service, performs read and write operations on multiple databases as normal. As shown in Figure 12.5, both databases A and C are locked.
2. When the application is about to commit the transaction, the coordinator asks all databases to prepare the transaction.
3. In the second phase, the coordinator collects replies from all databases and performs the following:
   (a) If all databases reply with a yes, the coordinator asks all databases to commit the transaction they have received.
   (b) If any database replies with a no, the coordinator asks all databases to abort the transaction.

It is a low-level solution because the prepare step requires a special modification to the database transaction. For example, there is an X/Open XA [6] standard that coordinates heterogeneous databases to achieve 2PC. The biggest problem with 2PC is that it's not performant, as locks can be held for a very long time while waiting for a message from the other nodes. Another issue with 2PC is that the coordinator can be a single point of failure, as shown in Figure 12.6.

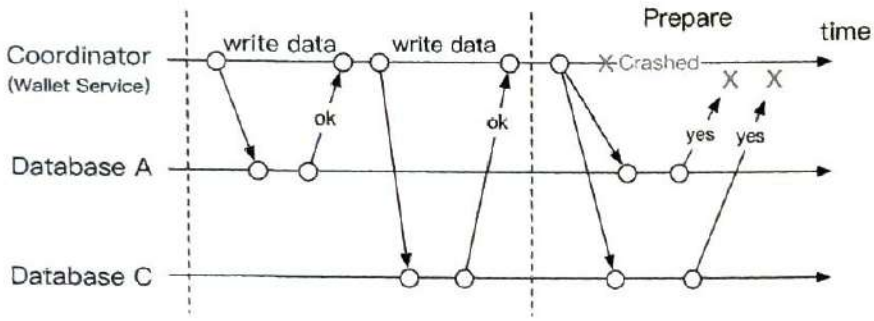


Figure 12.6: Coordinator crashes

Distributed transaction: Try-Confirm/Cancel (TC/C)

TC/C is a type of compensating transaction [7] that has two steps:

1. In the first phase, the coordinator asks all databases to reserve resources for the transaction.
2. In the second phase, the coordinator collects replies from all databases:
   (a) If all databases reply with yes, the coordinator asks all databases to confirm the operation, which is the Try-Confirm process.
   (b) If any database replies with no, the coordinator asks all databases to cancel the operation, which is the Try-Cancel process.

It's important to note that the two phases in 2PC are wrapped in the same transaction, but in TC/C each phase is a separate transaction.

TC/C example

It would be much easier to explain how TC/C works with a real-world example. Suppose we want to transfer $1 from account A to account C. Table 12.2 gives a summary of how TC/C is executed in each phase.

Phase	Operation	A	C
1	Try	Balance change: −$1	Do nothing
2	Confirm	Do nothing	Balance change: +$1
	Cancel	Balance change: +$1	Do Nothing

Table 12.2: TC/C example

Let's assume the wallet service is the coordinator of the TC/C. At the beginning of the distributed transaction, account A has $1 in its balance, and account C has $0.

First phase: Try In the Try phase, the wallet service, which acts as the coordinator, sends two transaction commands to two databases:

1. For the database that contains account A, the coordinator starts a local transaction that reduces the balance of A by $1.
2. For the database that contains account C, the coordinator gives it a `NOP` (no operation). To make the example adaptable for other scenarios, let's assume the coordinator sends to this database a `NOP` command. The database does nothing for `NOP` commands and always replies to the coordinator with a success message.

The Try phase is shown in Figure 12.7. The thick line indicates that a lock is held by the transaction.

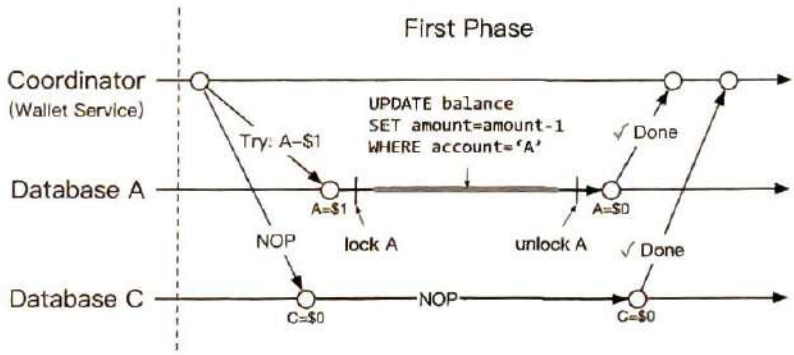


Figure 12.7: Try phase

Second phase: Confirm If both databases reply yes, the wallet service starts the next Confirm phase.

Account A's balance has already been updated in the first phase. The wallet service does not need to change its balance here. However, account C has not yet received its $1 from account A in the first phase. In the Confirm phase, the wallet service has to add $1 to account C's balance.

The Confirm process is shown in Figure 12.8.

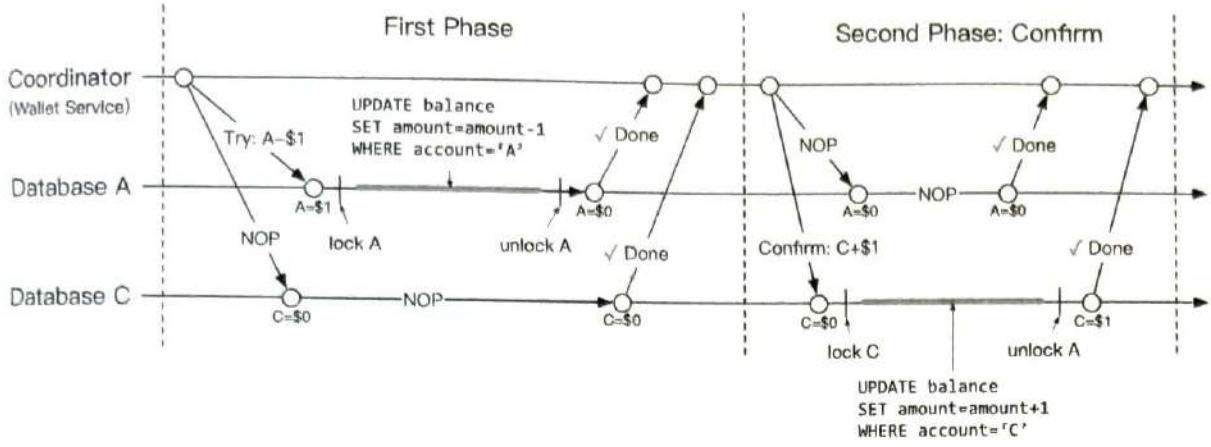


Figure 12.8: Confirm phase

Second phase: Cancel What if the first Try phase fails? In the example above we have assumed the NOP operation on account C always succeeds, although in practice it may fail. For example, account C might be an illegal account, and the regulator has mandated that no money can flow into or out of this account. In this case, the distributed transaction must be canceled and we have to clean up.

Because the balance of account A has already been updated in the transaction in the Try phase, it is impossible for the wallet service to cancel a completed transaction. What it can do is to start another transaction that reverts the effect of the transaction in the Try phase, which is to add $1 back to account A.

Because account C was not updated in the Try phase, the wallet service just needs to send a NOP operation to account C's database.

The Cancel process is shown in Figure 12.9.

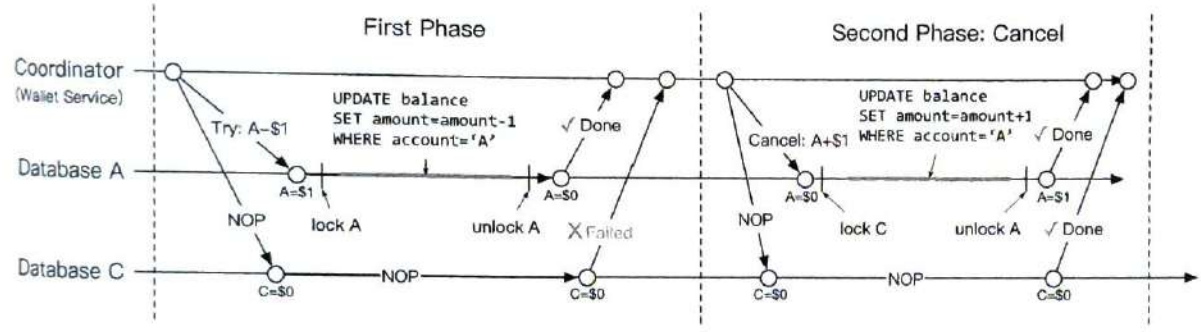


Figure 12.9: Cancel phase

Comparison between 2PC and TC/C

Table 12.3 shows that there are many similarities between 2PC and TC/C, but there are also differences. In 2PC, all local transactions are not done (still locked) when the second phase starts, while in TC/C, all local transactions are done (unlocked) when the second phase starts. In other words, the second phase of 2PC is about completing an unfinished

transaction, such as an abort or commit, while in TC/C, the second phase is about using a reverse operation to offset the previous transaction result when an error occurs. The following table summarizes their differences.

	First Phase	Second Phase: success	Second Phase: fail
2PC	Local transactions are not done yet	Commit all local transactions	Cancel all local transactions
TC/C	All local transactions are completed, either committed or canceled	Execute new local transactions if needed	Reverse the side effect of the already committed transaction, or called "undo"

Table 12.3: 2PC v.s. TC/C

TC/C is also called a distributed transaction by compensation. It is a high-level solution because the compensation, also called the "undo," is implemented in the business logic. The advantage of this approach is that it is database-agnostic. As long as a database supports transactions, TC/C will work. The disadvantage is that we have to manage the details and handle the complexity of the distributed transactions in the business logic at the application layer.

Phase status table

We still have not yet answered the question asked earlier; what if the wallet service restarts in the middle of TC/C? When it restarts, all previous operation history might be lost, and the system may not know how to recover.

The solution is simple. We can store the progress of a TC/C as phase status in a transactional database. The phase status includes at least the following information.

- The ID and content of a distributed transaction.
- The status of the Try phase for each database. The status could be `not sent yet`, `has been sent`, and `response received`.
- The name of the second phase. It could be `Confirm` or `Cancel`. It could be calculated using the result of the Try phase.
- The status of the second phase.
- An out-of-order flag (explained soon in the section "out-of-order Execution").

Where should we put the phase status tables? Usually, we store the phase status in the database that contains the wallet account from which money is deducted. The updated architecture diagram is shown in Figure 12.10.

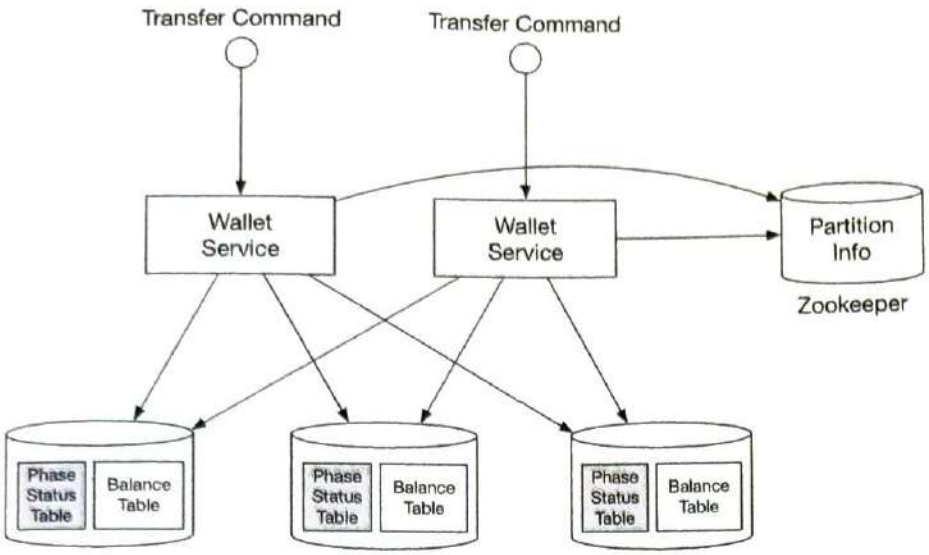


Figure 12.10: Phase status table

Unbalanced state

Have you noticed that by the end of the Try phase, $1 is missing (Figure 12.11)?

Assuming everything goes well, by the end of the Try phase, $1 is deducted from account A and account C remains unchanged. The sum of account balances in A and C will be $0, which is less than at the beginning of the TC/C. It violates a fundamental rule of accounting that the sum should remain the same after a transaction.

The good news is that the transactional guarantee is still maintained by TC/C. TC/C comprises several independent local transactions. Because TC/C is driven by application, the application itself is able to see the intermediate result between these local transactions. On the other hand, the database transaction or 2PC version of the distributed transaction was maintained by databases that are invisible to high-level applications.

There are always data discrepancies during the execution of distributed transactions. The discrepancies might be transparent to us because lower-level systems such as databases already fixed the discrepancies. If not, we have to handle it ourselves (for example, TC/C).

The unbalanced state is shown in Figure 12.11.

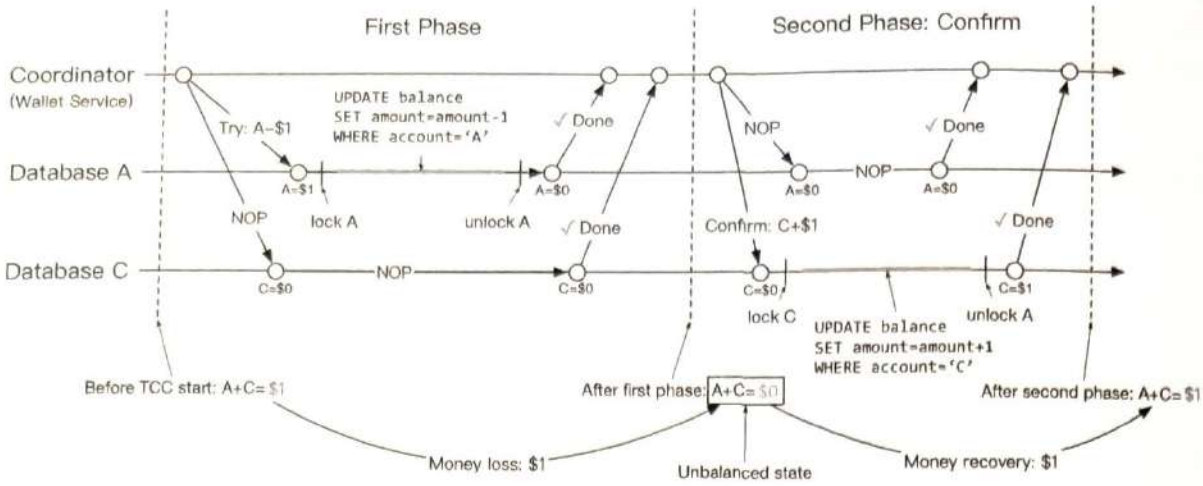


Figure 12.11: Unbalanced state

Valid operation orders

There are three choices for the Try phase:

Try phase choices	Account A	Account C
Choice 1	−$1	NOP
Choice 2	NOP	+$1
Choice 3	−$1	+$1

Table 12.4: Try phase choices

All three choices look plausible, but some are not valid.

For choice 2, if the Try phase on account C is successful, but has failed on account A (NOP), the wallet service needs to enter the Cancel phase. There is a chance that somebody else may jump in and move the $1 away from account C. Later when the wallet service tries to deduct $1 from account C, it finds nothing is left, which violates the transactional guarantee of a distributed transaction.

For choice 3, if $1 is deducted from account A and added to account C concurrently, it introduces lots of complications. For example, $1 is added to account C, but it fails to deduct the money from account A. What should we do in this case?

Therefore, choice 2 and choice 3 are flawed choices and only choice 1 is valid.

Out-of-order execution

One side effect of TC/C is the out-of-order execution. It will be much easier to explain using an example.

We reuse the above example which transfers $1 from account A to account C. As Figure 12.12 shows, in the Try phase, the operation against account A fails and it returns a failure to the wallet service, which then enters the Cancel phase and sends the cancel operation to both account A and account C.

Let's assume that the database that handles account C has some network issues and it

receives the Cancel instruction before the Try instruction. In this case, there is nothing to cancel.

The out-of-order execution is shown in Figure 12.12.

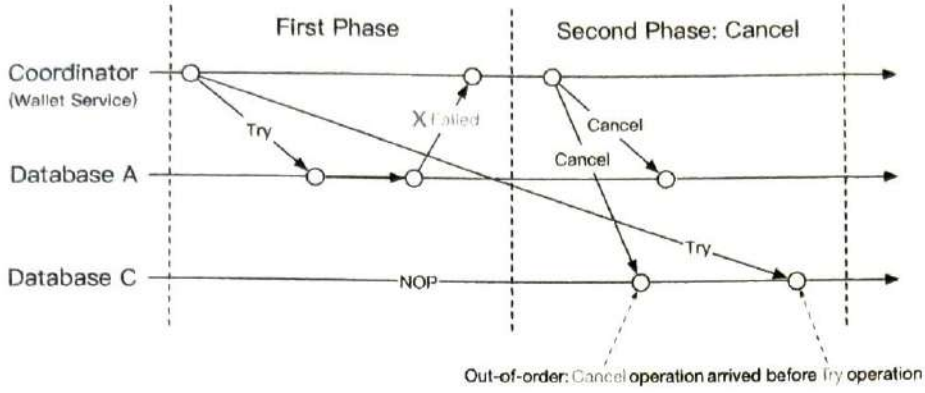


Figure 12.12: Out-of-order execution

To handle out-of-order operations, each node is allowed to Cancel a TC/C without receiving a Try instruction, by enhancing the existing logic with the following updates:

- The out-of-order Cancel operation leaves a flag in the database indicating that it has seen a Cancel operation, but it has not seen a Try operation yet.
- The Try operation is enhanced so it always checks whether there is an out-of-order flag, and it returns a failure if there is.

This is why we added an out-of-order flag to the phase status table in the "Phase Status Table" section.

Distributed transaction: Saga

Linear order execution

There is another popular distributed transaction solution called Saga [8]. Saga is the de-facto standard in a microservice architecture. The idea of Saga is simple:

1. All operations are ordered in a sequence. Each operation is an independent transaction on its own database.
2. Operations are executed from the first to the last. When one operation has finished, the next operation is triggered.
3. When an operation has failed, the entire process starts to roll back from the current operation to the first operation in reverse order, using compensating transactions. So if a distributed transaction has n operations, we need to prepare $2n$ operations: n operations for the normal case and another n for the compensating transaction during rollback.

It is easier to understand this by using an example. Figure 12.13 shows the Saga workflow to transfer $1 from account A to account C. The top horizontal line shows the normal

order of execution. The two vertical lines show what the system should do when there is an error. When it encounters an error, the transfer operations are rolled back and the client receives an error message. As we mentioned in the "Valid operation orders" section on page 352, we have to put the deduction operation before the addition operation.

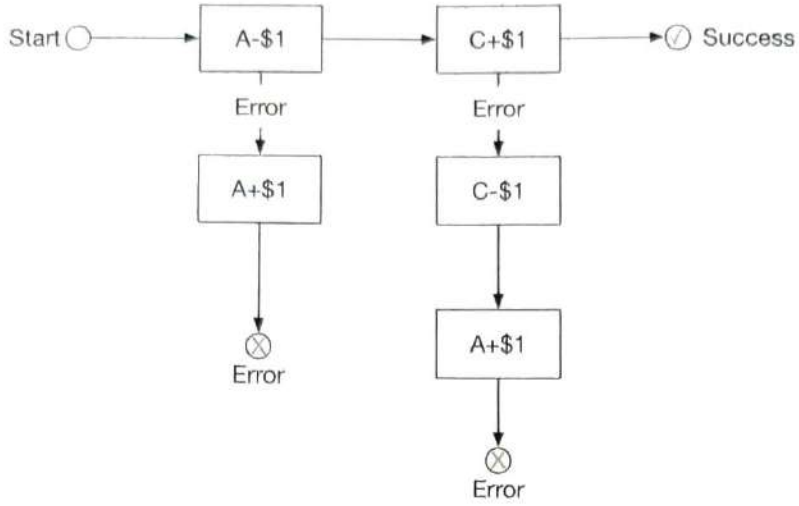


Figure 12.13: Saga workflow

How do we coordinate the operations? There are two ways to do it:

1. Choreography. In a microservice architecture, all the services involved in the Saga distributed transaction do their jobs by subscribing to other services' events. So it is fully decentralized coordination.
2. Orchestration. A single coordinator instructs all services to do their jobs in the correct order.

The choice of which coordination model to use is determined by the business needs and goals. The challenge of the choreography solution is that services communicate in a fully asynchronous way, so each service has to maintain an internal state machine in order to understand what to do when other services emit an event. It can become hard to manage when there are many services. The orchestration solution handles complexity well, so it is usually the preferred solution in a digital wallet system.

Comparison between TC/C and Saga

TC/C and Saga are both application-level distributed transactions. Table 12.5 summarizes their similarities and differences.

	TC/C	Saga
Compensating action	In Cancel phase	In rollback phase
Central coordination	Yes	Yes (orchestration mode)
Operation execution order	any	linear
Parallel execution possibility	Yes	No (linear execution)
Could see the partial inconsistent status	Yes	Yes
Application or database logic	Application	Application

Table 12.5: TC/C vs Saga

Which one should we use in practice? The answer depends on the latency requirement. As Table 12.5 shows, operations in Saga have to be executed in linear order, but it is possible to execute them in parallel in TC/C. So the decision depends on a few factors:

1. If there is no latency requirement, or there are very few services, such as our money transfer example, we can choose either of them. If we want to go with the trend in microservice architecture, choose Saga.
2. If the system is latency-sensitive and contains many services/operations, TC/C might be a better option.

Candidate: To make the balance transfer transactional, we replace Redis with a relational database, and use TC/C or Saga to implement distributed transactions.
Interviewer: Great work! The distributed transaction solution works, but there might be cases where it doesn't work well. For example, users might enter the wrong operations at the application level. In this case, the money we specified might be incorrect. We need a way to trace back the root cause of the issue and audit all account operations. How can we do this?

Event sourcing

Background

In real life, a digital wallet provider may be audited. These external auditors might ask some challenging questions, for example:

1. Do we know the account balance at any given time?
2. How do we know the historical and current account balances are correct?
3. How do we prove that the system logic is correct after a code change?

One design philosophy that systematically answers those questions is event sourcing, which is a technique developed in Domain-Driven Design (DDD) [9].

Definition

There are four important terms in event sourcing.

1. Command
2. Event
3. State
4. State machine

Command

A command is the intended action from the outside world. For example, if we want to transfer $1 from client A to client C, this money transfer request is a command.

In event sourcing, it is very important that everything has an order. So commands are usually put into a FIFO (first in, first out) queue.

Event

Command is an intention and not a fact because some commands may be invalid and cannot be fulfilled. For example, the transfer operation will fail if the account balance becomes negative after the transfer.

A command must be validated before we do anything about it. Once the command passes the validation, it is valid and must be fulfilled. The result of the fulfillment is called an event.

There are two major differences between command and event.

1. Events must be executed because they represent a validated fact. In practice, we usually use the past tense for an event. If the command is "transfer $1 from A to C", the corresponding event would be "transferred $1 from A to C".
2. Commands may contain randomness or I/O, but events must be deterministic. Events represent historical facts.

There are two important properties of the event generation process.

1. One command may generate any number of events. It could generate zero or more events.
2. Event generation may contain randomness, meaning it is not guaranteed that a command always generates the same event(s). The event generation may contain external I/O or random numbers. We will revisit this property in more detail near the end of the chapter.

The order of events must follow the order of commands. So events are stored in a FIFO queue, as well.

State

State is what will be changed when an event is applied. In the wallet system, state is the balances of all client accounts, which can be represented with a map data structure. The key is the account name or ID, and the value is the account balance. Key-value stores are usually used to store the map data structure. The relational database can also be viewed as a key-value store, where keys are primary keys and values are table rows.

State machine

A state machine drives the event sourcing process. It has two major functions.

1. Validate commands and generate events.
2. Apply event to update state.

Event sourcing requires the behavior of the state machine to be deterministic. Therefore, the state machine itself should never contain any randomness. For example, it should never read anything random from the outside using I/O, or use any random numbers. When it applies an event to a state, it should always generate the same result.

Figure 12.14 shows the static view of event sourcing architecture. The state machine is responsible for converting the command to an event and for applying the event. Because state machine has two primary functions, we usually draw two state machines, one for validating commands and the other for applying events.

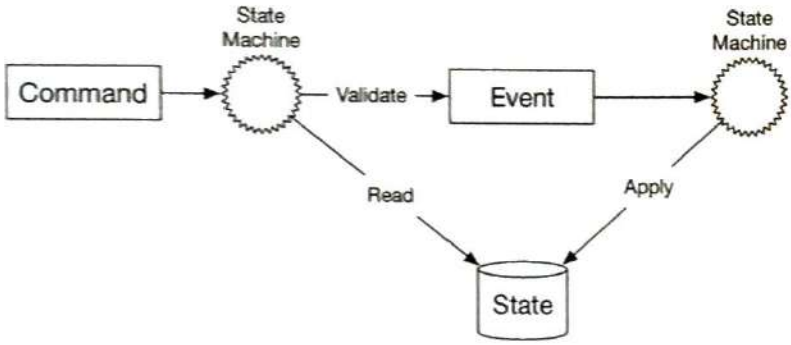


Figure 12.14: Static view of event sourcing

If we add the time dimension, Figure 12.15 shows the dynamic view of event sourcing. The system keeps receiving commands and processing them, one by one.

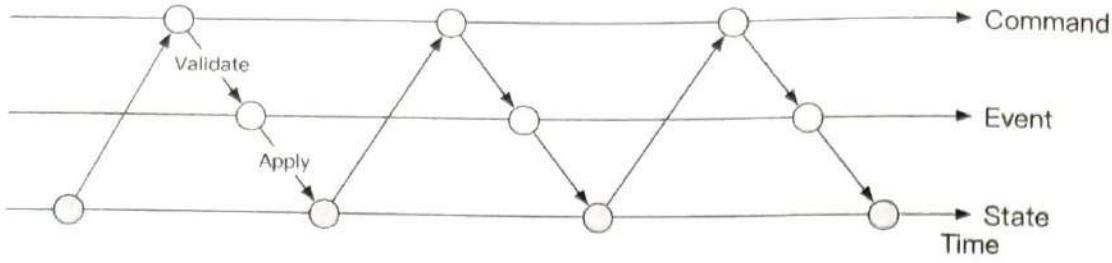


Figure 12.15: Dynamic view of event sourcing

Wallet service example

For the wallet service, the commands are balance transfer requests. These commands are put into a FIFO queue. One popular choice for the command queue is Kafka [10]. The command queue is shown in Figure 12.16.

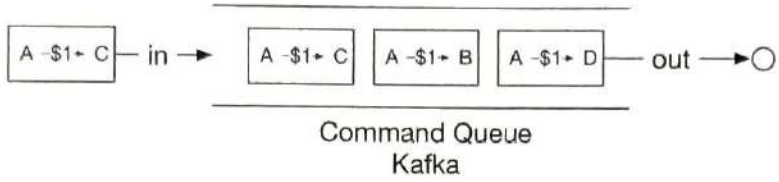


Figure 12.16: Command queue

Let us assume the state (the account balance) is stored in a relational database. The state machine examines each command one by one in FIFO order. For each command, it checks whether the account has a sufficient balance. If yes, the state machine generates an event for each account. For example, if the command is "A $\rightarrow$ \$1 $\rightarrow$ C", the state machine generates two events: "A:−\$1" and "C:+\$1".

Figure 12.17 shows how the state machine works in 5 steps.

1. Read commands from the command queue.
2. Read balance state from the database.
3. Validate the command. If it is valid, generate two events for each of the accounts.
4. Read the next event.
5. Apply the event by updating the balance in the database.

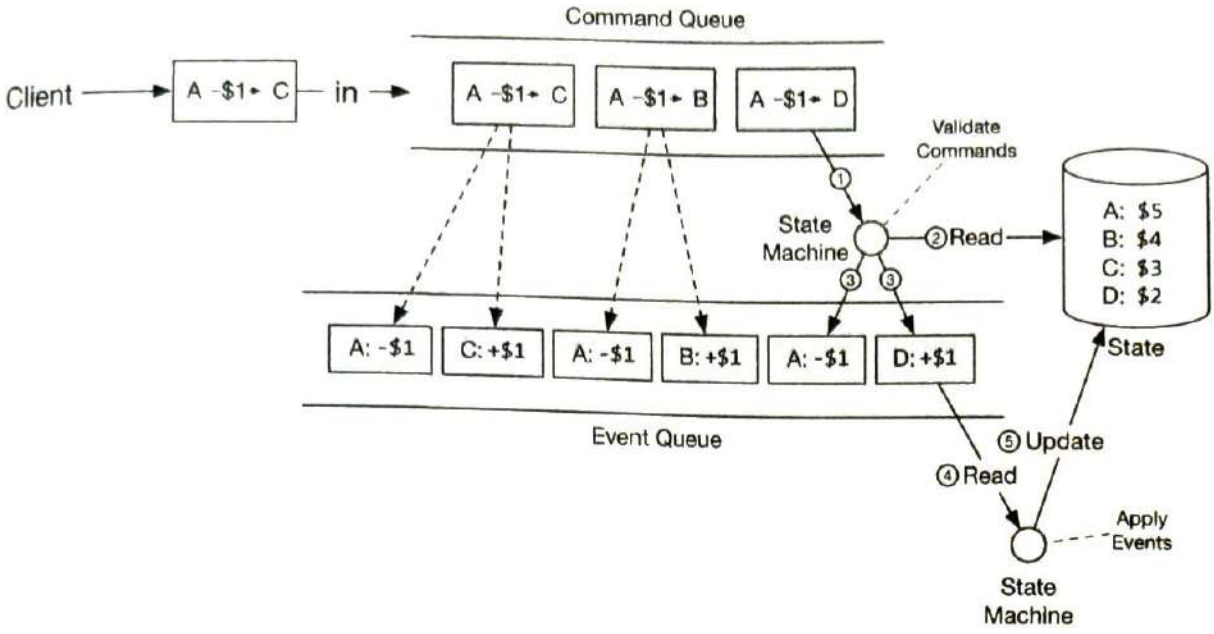


Figure 12.17: How state machine works

Reproducibility

The most important advantage that event sourcing has over other architectures is reproducibility.

In the distributed transaction solutions mentioned earlier, a wallet service saves the updated account balance (the state) into the database. It is difficult to know why the account balance was changed. Meanwhile, historical balance information is lost during the update operation. In the event sourcing design, all changes are saved first as immutable history. The database is only used as an updated view of what balance looks like at any given point in time.

We could always reconstruct historical balance states by replaying the events from the very beginning. Because the event list is immutable and the state machine logic is deterministic, it is guaranteed that the historical states generated from each replay are the same.

Figure 12.18 shows how to reproduce the states of the wallet service by replaying the events.

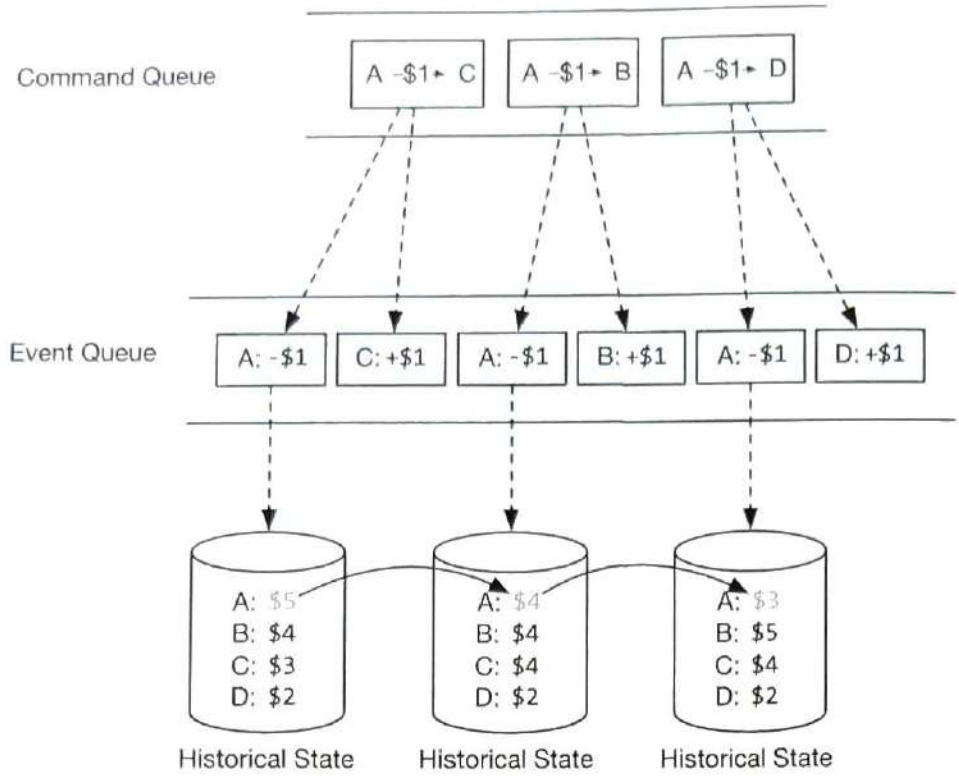


Figure 12.18: Reproduce states

Reproducibility helps us answer the difficult questions that the auditors ask at the beginning of the section. We repeat the questions here.

1. Do we know the account balance at any given time?
2. How do we know the historical and current account balances are correct?
3. How do we prove the system logic is correct after a code change?

For the first question, we could answer it by replaying events from the start, up to the point in time where we would like to know the account balance.

For the second question, we could verify the correctness of the account balance by recalculating it from the event list.

For the third question, we can run different versions of the code against the events and verify that their results are identical.

Because of the audit capability, event sourcing is often chosen as the de facto solution for the wallet service.

Command-query responsibility segregation (CQRS)

So far, we have designed the wallet service to move money from one account to another efficiently. However, the client still does not know what the account balance is. There needs to be a way to publish state (balance information) so the client, which is outside of the event sourcing framework, can know what the state is.

Intuitively, we can create a read-only copy of the database (historical state) and share

it with the outside world. Event sourcing answers this question in a slightly different way.

Rather than publishing the state (balance information), event sourcing publishes all the events. The external world could rebuild any customized state itself. This design philosophy is called CQRS [11].

In CQRS, there is one state machine responsible for the write part of the state, but there can be many read-only state machines, which are responsible for building views of the states. Those views could be used for queries.

These read-only state machines can derive different state representations from the event queue. For example, clients may want to know their balances and a read-only state machine could save state in a database to serve the balance query. Another state machine could build state for a specific time period to help investigate issues like possible double charges. The state information is an audit trail that could help to reconcile the financial records.

The read-only state machines lag behind to some extent, but will always catch up. The architecture design is eventually consistent.

Figure 12.19 shows a classic CQRS architecture.

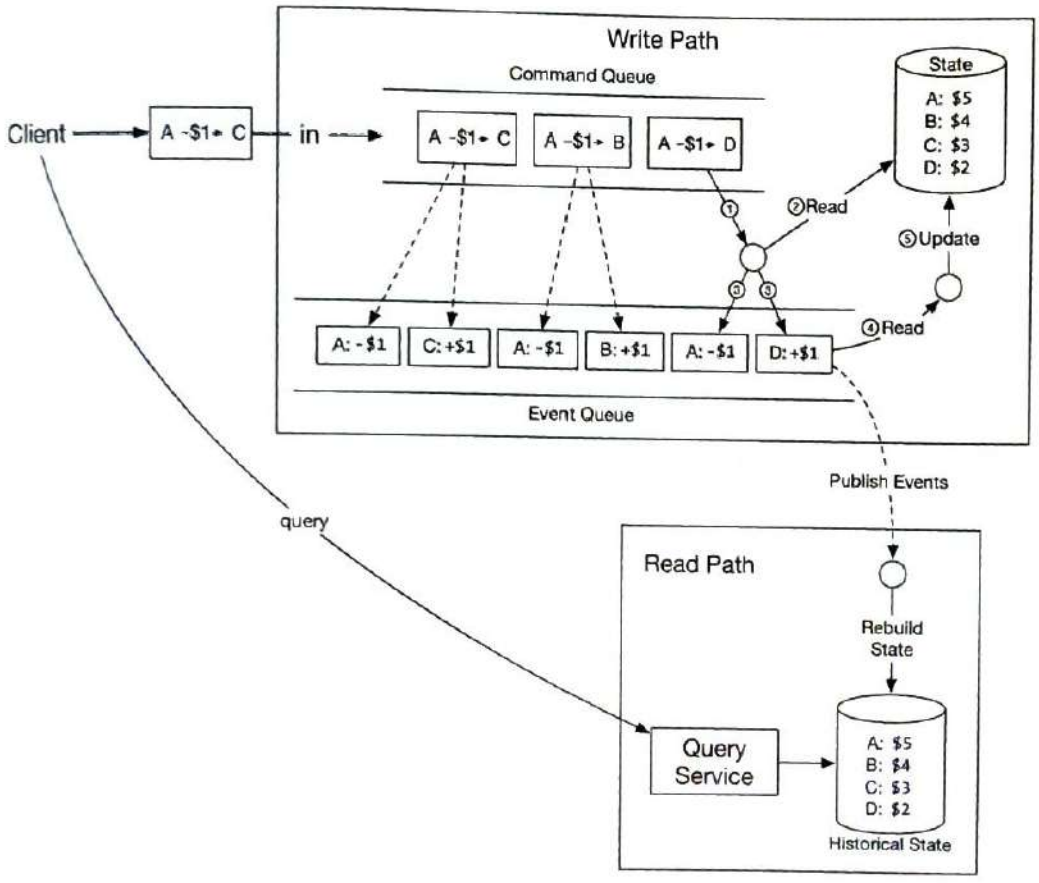


Figure 12.19: CQRS architecture

Candidate: In this design, we use event sourcing architecture to make the whole system reproducible. All valid business records are saved in an immutable event queue which could be used for correctness verification.

Interviewer: That's great. But the event sourcing architecture you proposed only handles one event at a time and it needs to communicate with several external systems. Can we make it faster?

Step 3 - Design Deep Dive

In this section, we dive deep into techniques for achieving high performance, reliability, and scalability.

High-performance event sourcing

In the earlier example, we used Kafka as the command and event store, and the database as a state store. Let's explore some optimizations.

File-based command and event list

The first optimization is to save commands and events to a local disk, rather than to a remote store like Kafka. This avoids transit time across the network. The event list uses an append-only data structure. Appending is a sequential write operation, which is generally very fast. It works well even for magnetic hard drives because the operating system is heavily optimized for sequential reads and writes. According to this article [12], sequential disk access can be faster than random memory access in some cases.

The second optimization is to cache recent commands and events in memory. As we explained before, we process commands and events right after they are persisted. We may cache them in memory to save the time of loading them back from the local disk.

We are going to explore some implementation details. A technique called mmap [13] is great for implementing the optimizations mentioned previously. Mmap can write to a local disk and cache recent content in memory at the same time. It maps a disk file to memory as an array. The operating system caches certain sections of the file in memory to accelerate the read and write operations. For append-only file operations, it is almost guaranteed that all data are saved in memory, which is very fast.

Figure 12.20 shows the file-based command and event storage.

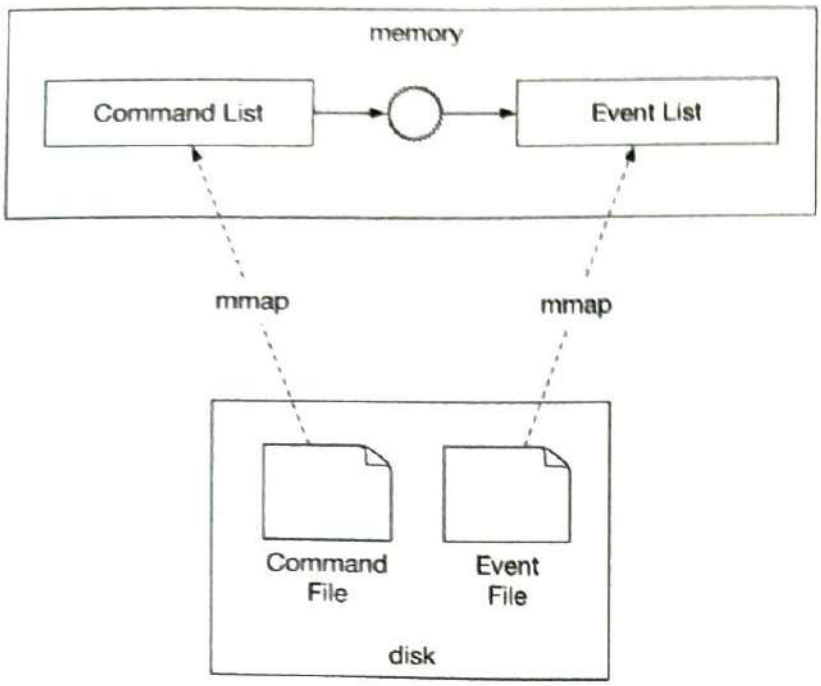


Figure 12.20: File-based command and event storage

File-based state

In the previous design, state (balance information) is stored in a relational database. In a production environment, a database usually runs in a stand-alone server that can only be accessed through networks. Similar to the optimizations we did for command and event, state information can be saved to the local disk, as well.

More specifically, we can use SQLite [14], which is a file-based local relational database or use RocksDB [15], which is a local file-based key-value store.

RocksDB is chosen because it uses a log-structured merge-tree (LSM), which is optimized for write operations. To improve read performance, the most recent data is cached.

Figure 12.21 shows the file-based solution for command, event, and state.

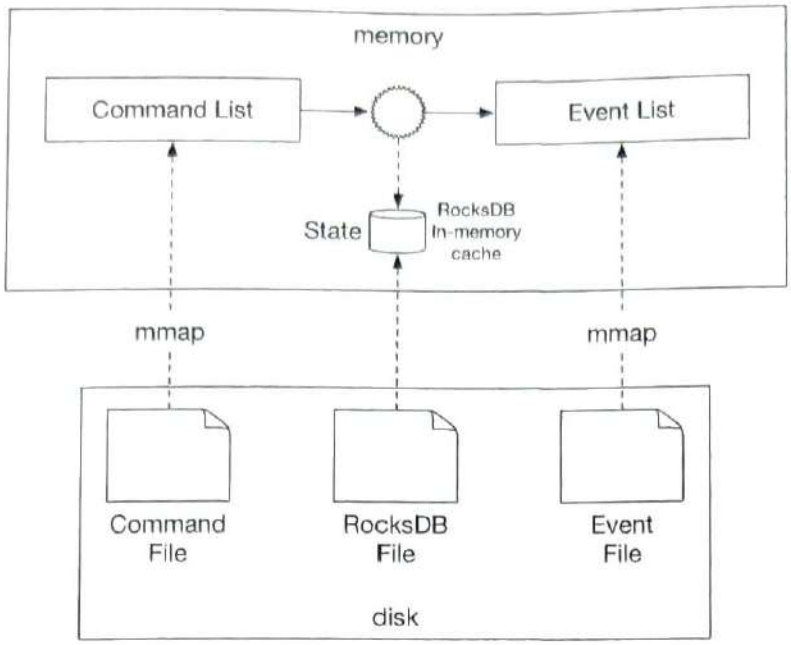


Figure 12.21: File-based solution for command, event, and state

Snapshot

Once everything is file-based, let us consider how to accelerate the reproducibility process. When we first introduced reproducibility, the state machine had to process events from the very beginning, every time. What we could optimize is to periodically stop the state machine and save the current state into a file. This is called a snapshot.

A snapshot is an immutable view of a historical state. Once a snapshot is saved, the state machine does not have to restart from the very beginning anymore. It can read data from a snapshot, verify where it left off, and resume processing from there.

For financial applications such as wallet service, the finance team often requires a snapshot to be taken at 00:00 so they can verify all transactions that happened during that day. When we first introduced CQRS of event sourcing, the solution was to set up a read-only state machine that reads from the beginning until the specified time is met. With snapshots, a read-only state machine only needs to load one snapshot that contains the data.

A snapshot is a giant binary file and a common solution is to save it in an object storage solution, such as HDFS [16].

Figure 12.22 shows the file-based event sourcing architecture. When everything is file-based, the system can fully utilize the maximum I/O throughput of the computer hardware.

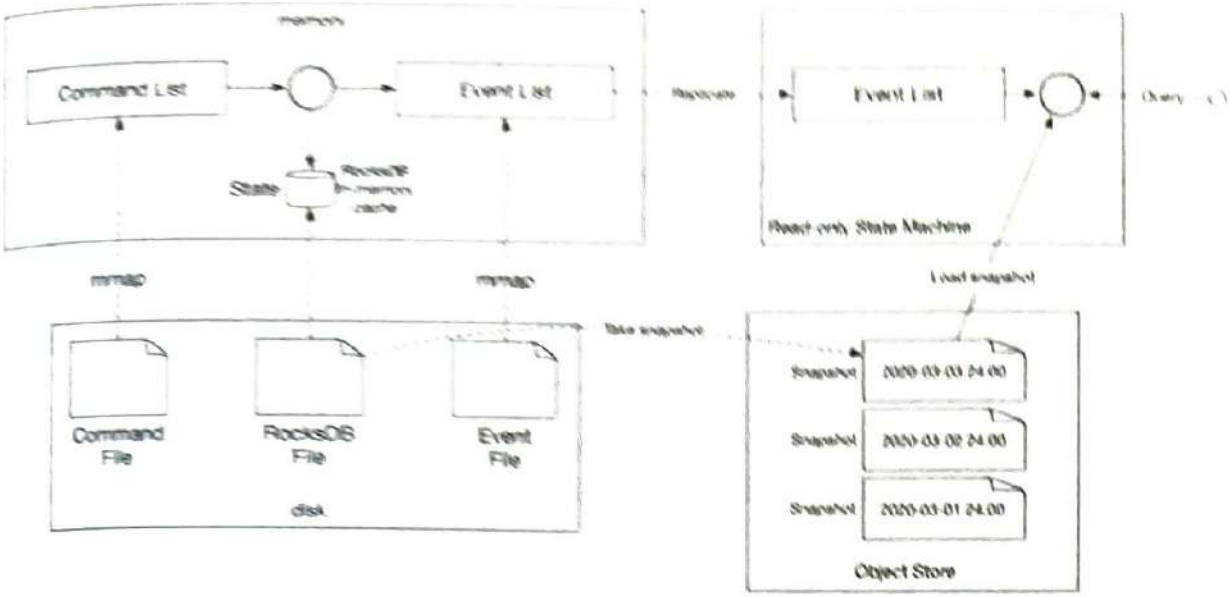


Figure 12.22: Snapshot

Candidate: We could refactor the design of event sourcing so the command list, event list, state, and snapshot are all saved in files. Event sourcing architecture processes the event list in a linear manner, which fits well into the design of hard disks and operating system cache.

Interviewer: The performance of the local file-based solution is better than the system that requires accessing data from remote Kafka and databases. However, there is another problem: because data is saved on a local disk, a server is now stateful and becomes a single point of failure. How do we improve the reliability of the system?

Reliable high-performance event sourcing

Before we explain the solution, let's examine the parts of the system that need the reliability guarantee.

Reliability analysis

Conceptually, everything a node does is around two concepts; data and computation. As long as data is durable, it's easy to recover the computational result by running the same code on another node. This means we only need to worry about the reliability of data because if data is lost, it is lost forever. The reliability of the system is mostly about the reliability of the data.

There are four types of data in our system.

1. File-based command
2. File-based event
3. File-based state
4. State snapshot

Let us take a close look at how to ensure the reliability of each type of data.

State and snapshot can always be regenerated by replaying the event list. To improve the reliability of state and snapshot, we just need to ensure the event list has strong reliability.

Now let us examine command. On the face of it, event is generated from command. We might think providing a strong reliability guarantee for command should be sufficient. This seems to be correct at first glance, but it misses something important. Event generation is not guaranteed to be deterministic, and also it may contain random factors such as random numbers, external I/O, etc. So command cannot guarantee reproducibility of events.

Now it's time to take a close look at event. Event represents historical facts that introduce changes to the state (account balance). Event is immutable and can be used to rebuild the state.

From this analysis, we conclude that event data is the only one that requires a high-reliability guarantee. We will explain how to achieve this in the next section.

Consensus

To provide high reliability, we need to replicate the event list across multiple nodes. During the replication process, we have to guarantee the following properties.

1. No data loss.
2. The relative order of data within a log file remains the same across nodes.

To achieve those guarantees, consensus-based replication is a good fit. The consensus algorithm makes sure that multiple nodes reach a consensus on what the event list is. Let's use the Raft [17] consensus algorithm as an example.

The Raft algorithm guarantees that as long as more than half of the nodes are online, the append-only lists on them have the same data. For example, if we have 5 nodes and use the Raft algorithm to synchronize their data, as long as at least 3 (more than $\frac{1}{2}$) of the nodes are up as Figure 12.23 shows, the system can still work properly as a whole:

Figure 12.23: Raft

A node can have three different roles in the Raft algorithm.

1. Leader
2. Candidate
3. Follower

We can find the implementation of the Raft algorithm in the Raft paper. We will only cover the high level concepts here and not go into detail. In Raft, at most one node is the leader of the cluster and the remaining nodes are followers. The leader is respon-

sible for receiving external commands and replicating data reliably across nodes in the cluster.

With the Raft algorithm, the system is reliable as long as the majority of the nodes are operational. For example, if there are 3 nodes in the cluster, it could tolerate the failure of 1 node, and if there are 5 nodes, it can tolerate the failure of 2 nodes.

Reliable solution

With replication, there won't be a single point of failure in our file-based event sourcing architecture. Let's take a look at the implementation details. Figure 12.24 shows the event sourcing architecture with the reliability guarantee.

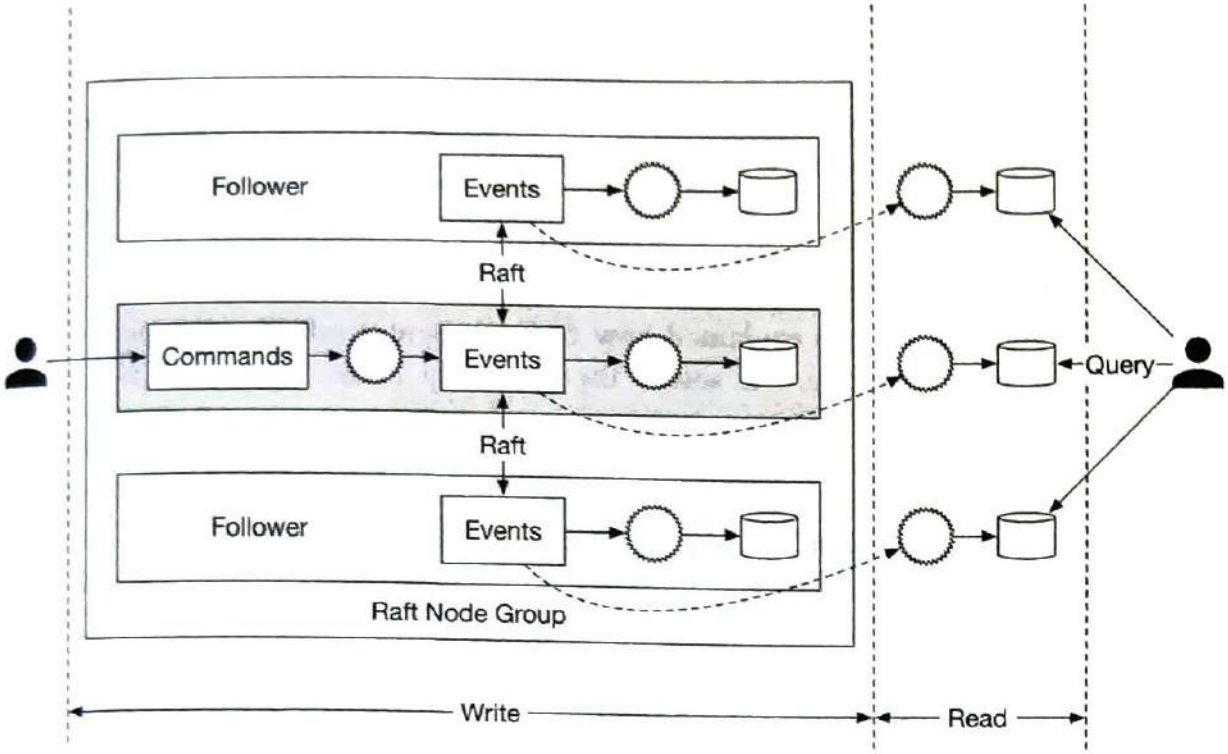


Figure 12.24: Raft node group

In Figure 12.24, we set up 3 event sourcing nodes. These nodes use the Raft algorithm to synchronize the event list reliably.

The leader takes incoming command requests from external users, converts them into events, and appends events into the local event list. The Raft algorithm replicates newly added events to the followers.

All nodes, including the followers, process the event list and update the state. The Raft algorithm ensures the leader and followers have the same event lists, while event sourcing guarantees all states are the same, as long as the event lists are the same.

A reliable system needs to handle failures gracefully, so let's explore how node crashes are handled.

If the leader crashes, the Raft algorithm automatically selects a new leader from the remaining healthy nodes. This newly elected leader takes responsibility for accepting

commands from external users. It is guaranteed that the cluster as a whole can provide continued service when a node goes down.

When the leader crashes, it is possible that the crash happens before the command list is converted to events. In this case, the client would notice the issue either by a timeout or by receiving an error response. The client needs to resend the same command to the newly elected leader.

In contrast, follower crashes are much easier to handle. If a follower crashes, requests sent to it will fail. Raft handles failures by retrying indefinitely until the crashed node is restarted or a new one replaces it.

Candidate: In this design, we use the Raft consensus algorithm to replicate the event list across multiple nodes. The leader receives commands and replicates events to other nodes.

Interviewer: Yes, the system is more reliable and fault-tolerant. However, in order to handle 1 million TPS, one server is not enough. How can we make the system more scalable?

Distributed event sourcing

In the previous section, we explained how to implement a reliable high-performance event sourcing architecture. It solves the reliability issue, but it has two limitations.

1. When a digital wallet is updated, we want to receive the updated result immediately. But in the CQRS design, the request/response flow can be slow. This is because a client doesn't know exactly when a digital wallet is updated and the client may need to rely on periodic polling.
2. The capacity of a single Raft group is limited. At a certain scale, we need to shard the data and implement distributed transactions.

Let's take a look at how those two problems are solved.

Pull vs push

In the pull model, an external user periodically polls execution status from the read-only state machine. This model is not real-time and may overload the wallet service if the polling frequency is set too high. Figure 12.25 shows the pulling model.

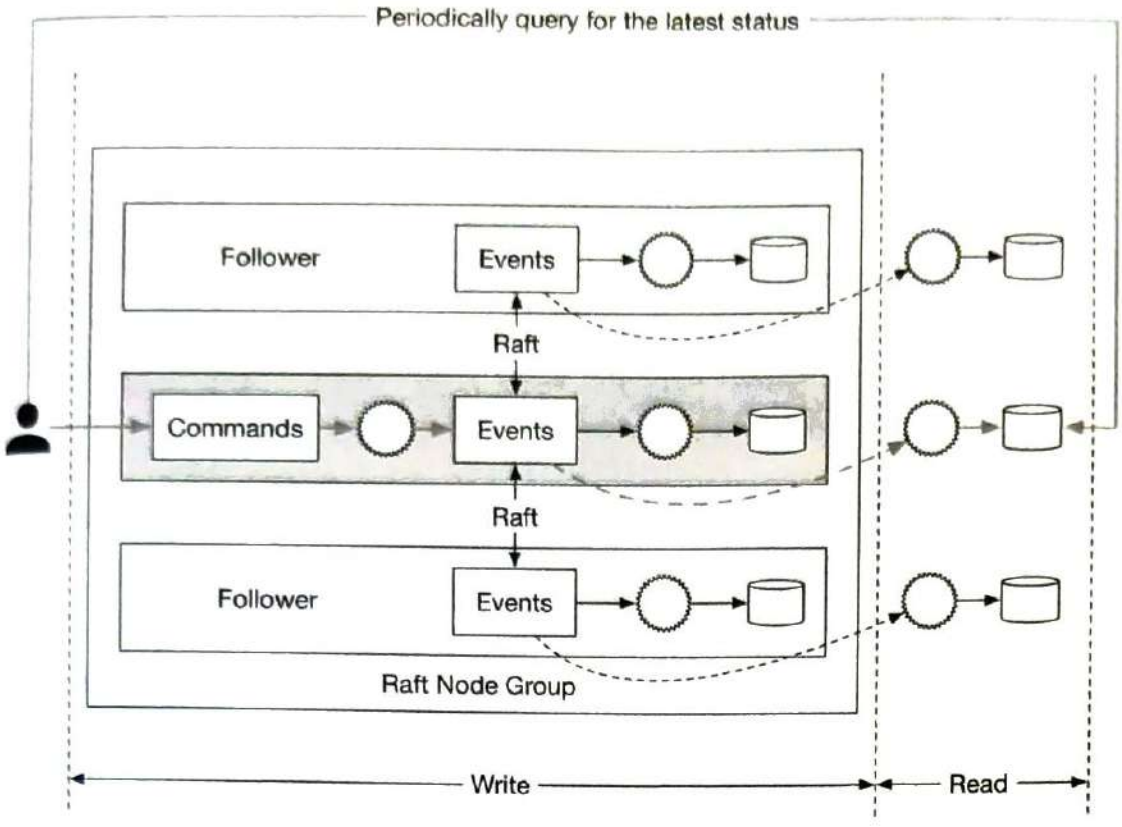


Figure 12.25: Periodical pulling

The naive pull model can be improved by adding a reverse proxy [18] between the external user and the event sourcing node. In this design, the external user sends a command to the reverse proxy, which forwards the command to event sourcing nodes and periodically polls the execution status. This design simplifies the client logic, but the communication is still not real-time.

Figure 12.26 shows the pull model with a reverse proxy added.

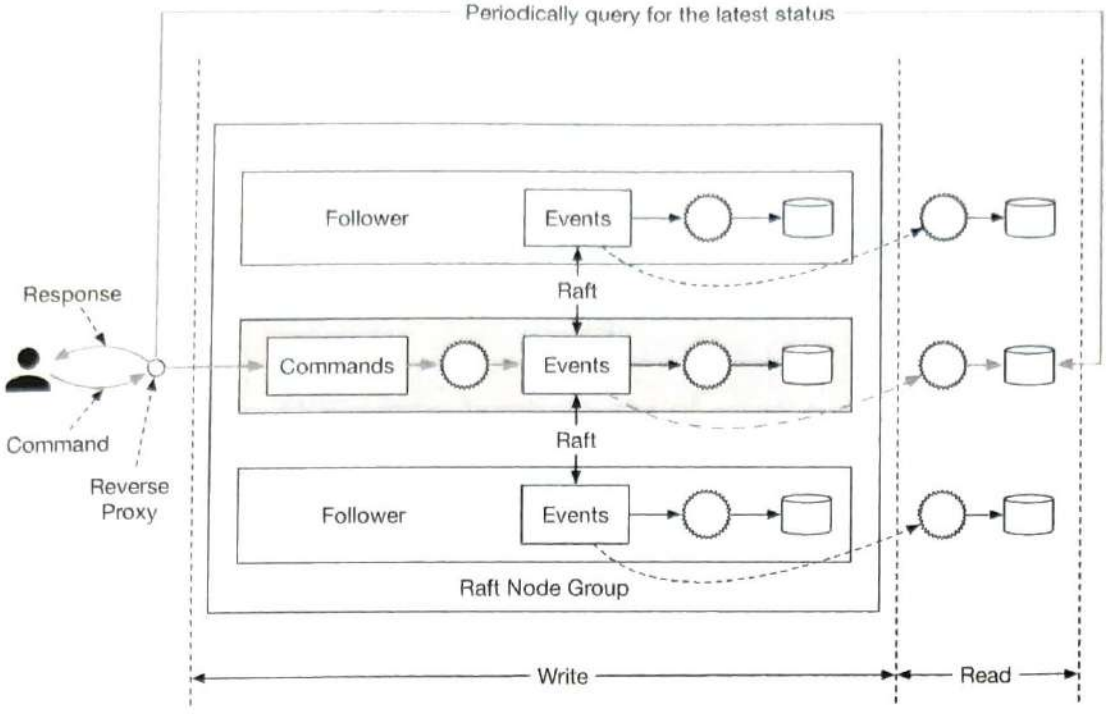


Figure 12.26: Pull model with reverse proxy

Once we have the reverse proxy, we could make the response faster by modifying the read-only state machine. As we mentioned earlier, the read-only state machine could have its own behavior. For example, one behavior could be that the read-only state machine pushes execution status back to the reverse proxy, as soon as it receives the event. This will give the user a feeling of real-time response.

Figure 12.27 shows the push-based model.

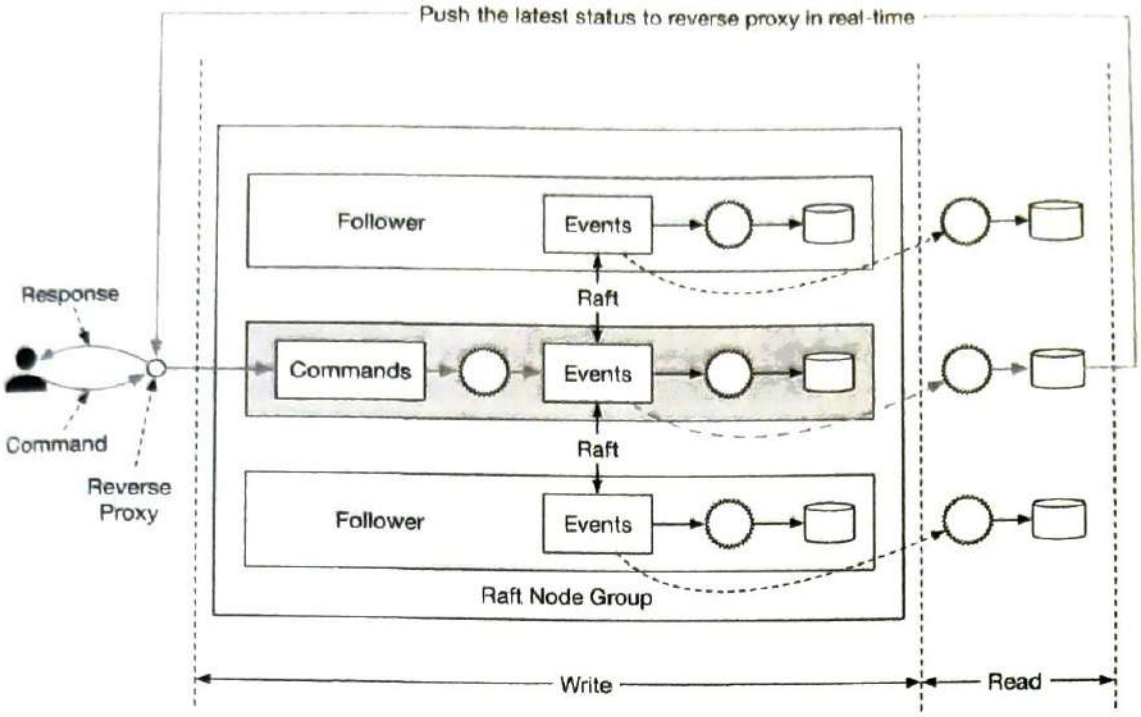


Figure 12.27: Push model

Distributed transaction

Once synchronous execution is adopted for every event sourcing node group, we can reuse the distributed transaction solution, TC/C or Saga. Assume we partition the data by dividing the hash value of keys by 2.

Figure 12.28 shows the updated design.

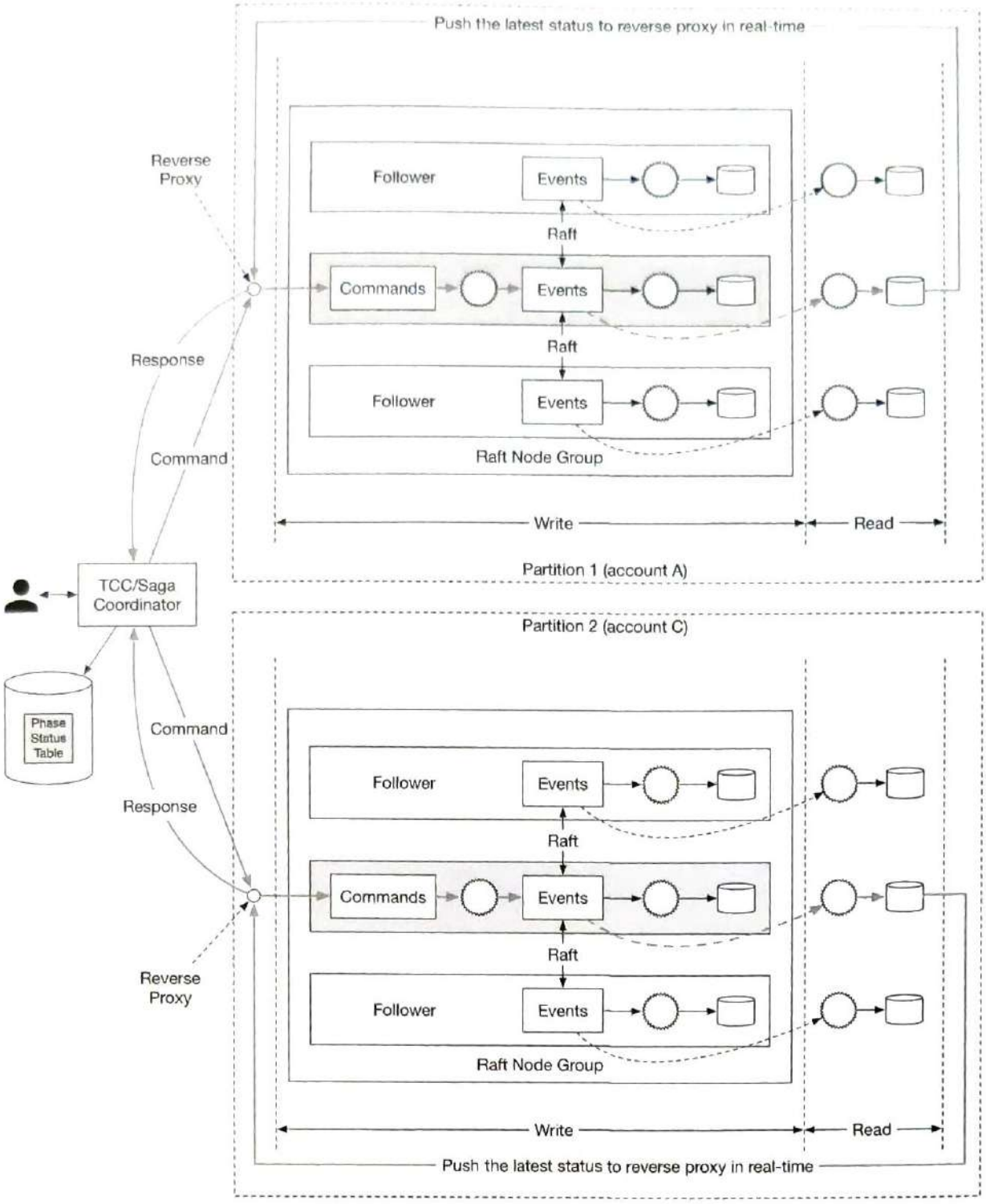


Figure 12.28: Final design

Let's take a look at how the money transfer works in the final distributed event sourcing architecture. To make it easier to understand, we use the Saga distributed transaction model and only explain the happy path without any rollback.

The money transfer operation contains 2 distributed operations: A:−$1 and C:+$1. The Saga coordinator coordinates the execution as shown in Figure 12.29:

1. User A sends a distributed transaction to the Saga coordinator. It contains two operations: A:−$1 and C:+$1.
2. Saga coordinator creates a record in the phase status table to trace the status of a transaction.
3. Saga coordinator examines the order of operations and determines that it needs to handle A:−$1 first. The coordinator sends A:−$1 as a command to Partition 1, which contains account A's information.
4. Partition 1's Raft leader receives the A−$1 command and stores it in the command list. It then validates the command. If it is valid, it is converted into an event. The Raft consensus algorithm is used to synchronize data across different nodes. The event (deducting $1 from A's account balance) is executed after synchronization is complete.
5. After the event is synchronized, the event sourcing framework of Partition 1 synchronizes the data to the read path using CQRS. The read path reconstructs the state and the status of execution.
6. The read path of Partition 1 pushes the status back to the caller of the event sourcing framework, which is the Saga coordinator.
7. Saga coordinator receives the success status from Partition 1.
8. The Saga coordinator creates a record, indicating the operation in Partition 1 is successful, in the phase status table.
9. Because the first operation succeeds, the Saga coordinator executes the second operation, which is C:+$1. The coordinator sends C:+$1 as a command to Partition 2 which contains account C's information.
10. Partition 2's Raft leader receives the C:+$1 command and saves it to the command list. If it is valid, it is converted into an event. The Raft consensus algorithm is used to synchronize data across different nodes. The event (add $1 to C's account) is executed after synchronization is complete.
11. After the event is synchronized, the event sourcing framework of Partition 2 synchronizes the data to the read path using CQRS. The read path reconstructs the state and the status of execution.
12. The read path of Partition 2 pushes the status back to the caller of the event sourcing framework, which is the Saga coordinator.
13. The Saga coordinator receives the success status from Partition 2.
14. The Saga coordinator creates a record, indicating the operation in Partition 2 is successful in the phase status table.
15. At this time, all operations succeed and the distributed transaction is completed. The Saga coordinator responds to its caller with the result.

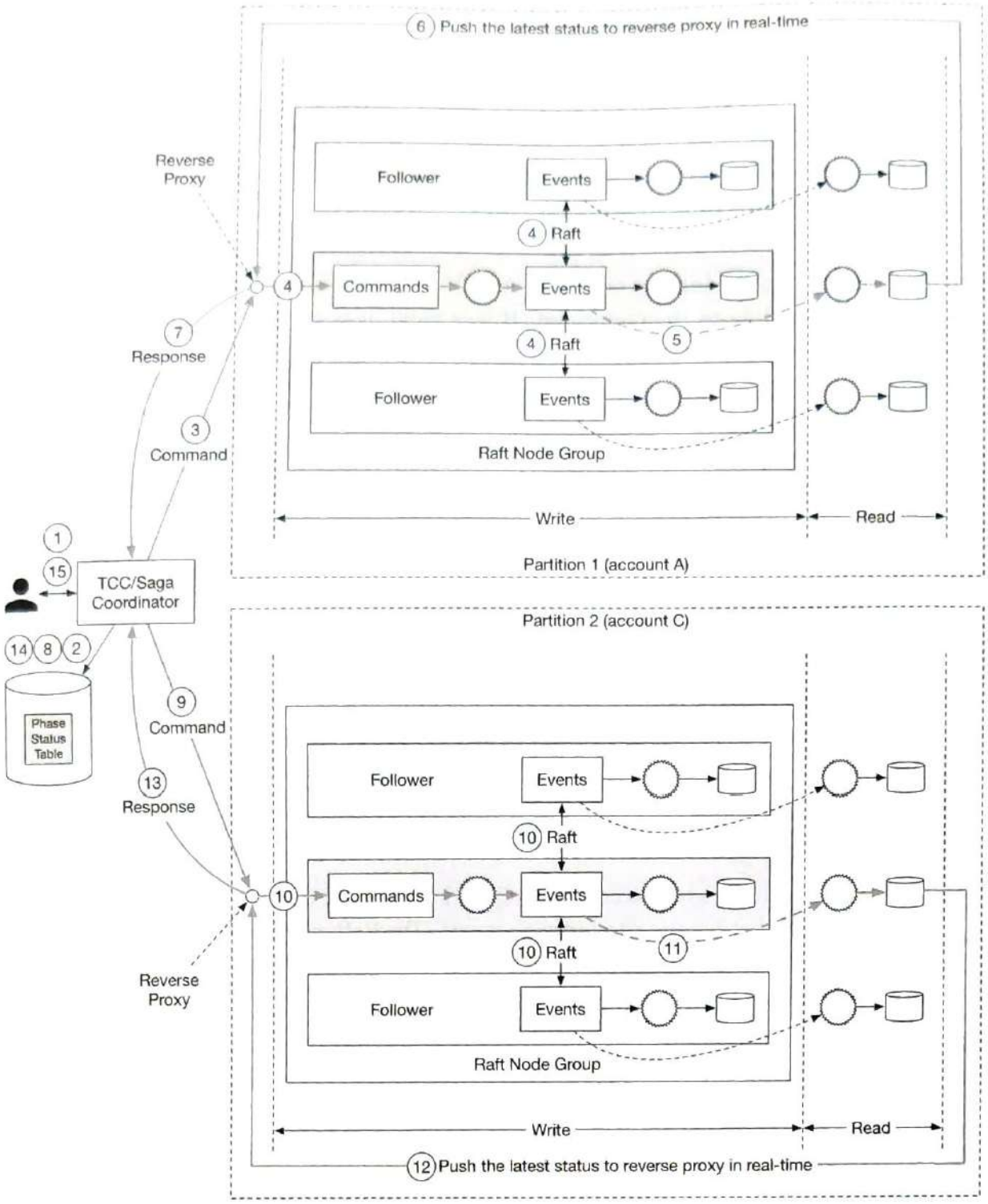


Figure 12.29: Final design in a numbered sequence

Step 4 - Wrap Up

In this chapter, we designed a wallet service that is capable of processing over 1 million payment commands per second. After a back-of-the-envelope estimation, we concluded that a few thousand nodes are required to support such a load.

In the first design, a solution using in-memory key-value stores like Redis is proposed. The problem with this design is that data isn't durable.

In the second design, the in-memory cache is replaced by transactional databases. To support multiple nodes, different transactional protocols such as 2PC, TC/C, and Saga are proposed. The main issue with transaction-based solutions is that we cannot conduct a data audit easily.

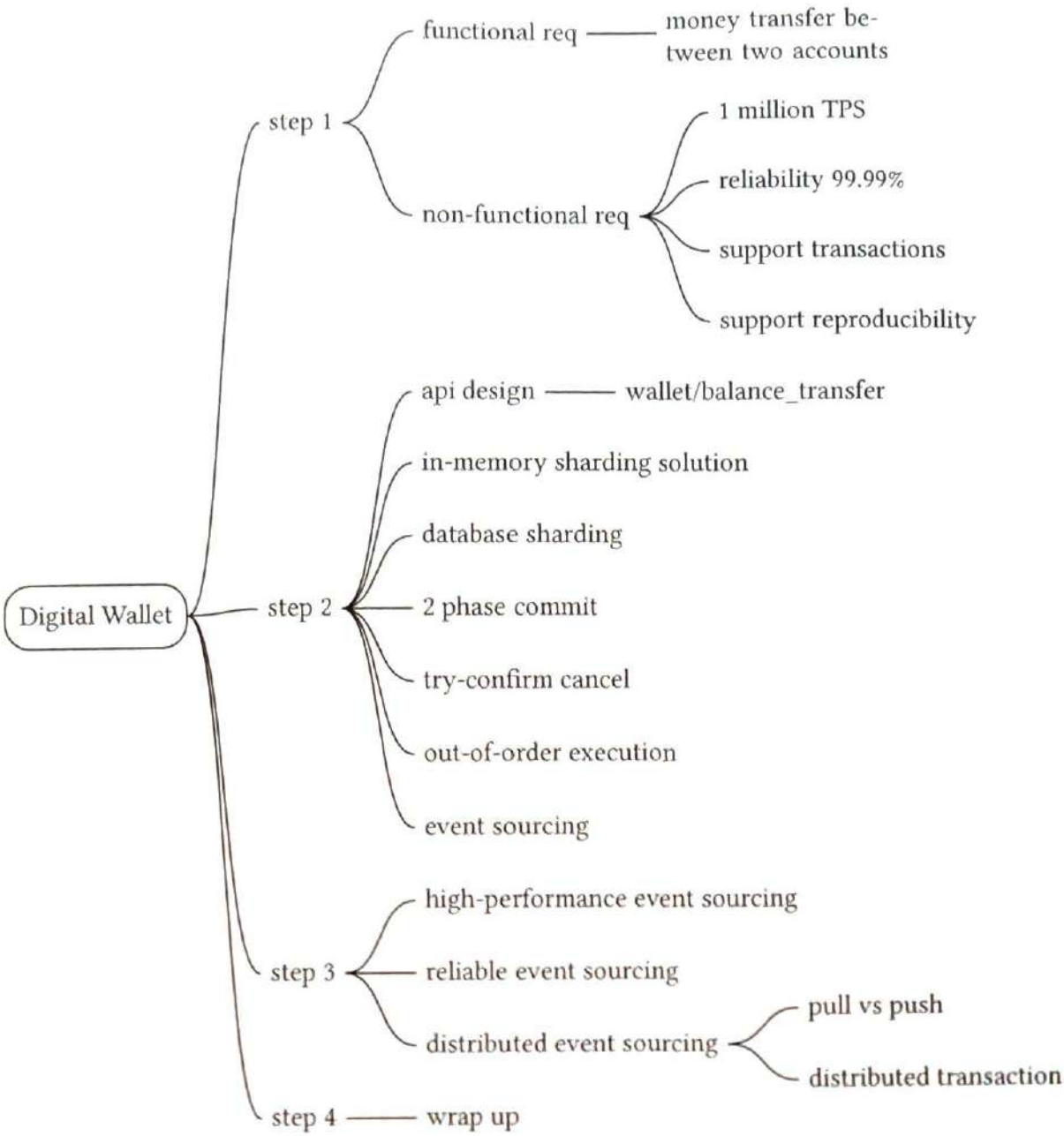
Next, event sourcing is introduced. We first implemented event sourcing using an external database and queue, but it's not performant. We improved performance by storing command, event, and state in a local node.

A single node means a single point of failure. To increase the system reliability, we use the Raft consensus algorithm to replicate the event list onto multiple nodes.

The last enhancement we made was to adopt the CQRS feature of event sourcing. We added a reverse proxy to change the asynchronous event sourcing framework to a synchronous one for external users. The TC/C or Saga protocol is used to coordinate Command executions across multiple node groups.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Chapter Summary



Reference Material

[1] Transactional guarantees. https://docs.oracle.com/cd/E17275_01/html/programmer_reference/rep_trans.html.

[2] TPC-E Top Price/Performance Results. http://tpc.org/tpce/results/tpce_price_perf_results5.asp?resulttype=all.

[3] ISO 4217 CURRENCY CODES. https://en.wikipedia.org/wiki/ISO_4217.

[4] Apache ZooKeeper. https://zookeeper.apache.org/.

[5] Martin Kleppmann. *Designing Data-Intensive Applications*. O'Reilly Media, 2017.

[6] X/Open XA. https://en.wikipedia.org/wiki/X/Open_XA.

[7] Compensating transaction. https://en.wikipedia.org/wiki/Compensating_transaction.

[8] SAGAS, HectorGarcia-Molina. https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf.

[9] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional, 2003.

[10] Apache Kafka. https://kafka.apache.org/.

[11] CQRS. https://martinfowler.com/bliki/CQRS.html.

[12] Comparing Random and Sequential Access in Disk and Memory. https://deliveryimages.acm.org/10.1145/1570000/1563874/jacobs3.jpg.

[13] mmap. https://man7.org/linux/man-pages/man2/mmap.2.html.

[14] SQLite. https://www.sqlite.org/index.html.

[15] RocksDB. https://rocksdb.org/.

[16] Apache Hadoop. https://hadoop.apache.org/.

[17] Raft. https://raft.github.io/.

[18] Reverse proxy. https://en.wikipedia.org/wiki/Reverse_proxy.

13 Stock Exchange

In this chapter, we design an electronic stock exchange system.

The basic function of an exchange is to facilitate the matching of buyers and sellers efficiently. This fundamental function has not changed over time. Before the rise of computing, people exchanged tangible goods by bartering and shouting at each other to get matched. Today, orders are processed silently by supercomputers, and people trade not only for the exchange of products, but also for speculation and arbitrage. Technology has greatly changed the landscape of trading and exponentially boosted electronic market trading volume.

When it comes to stock exchanges, most people think about major market players like The New York Stock exchange (NYSE) or Nasdaq, which have existed for over fifty years. In fact, there are many other types of exchange. Some focus on vertical segmentation of the financial industry and place special focus on technology [1], while others have an emphasis on fairness [2]. Before diving into the design, it is important to check with the interviewer about the scale and the important characteristics of the exchange in question.

Just to get a taste of the kind of problem we are dealing with; NYSE is trading billions of matches per day [3], and HKEX about 200 billion shares per day [4]. Figure 13.1 shows the big exchanges in the "trillion-dollar club" by market capitalization.

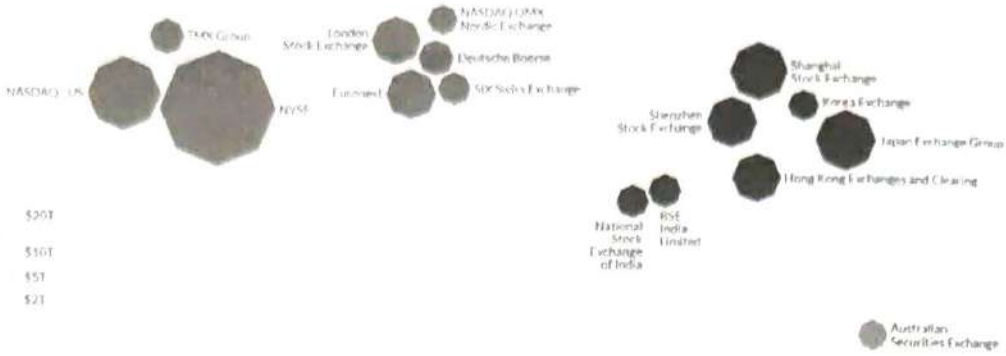


Figure 13.1: Largest stock exchanges (Source: [5])

Step 1 - Understand the Problem and Establish Design Scope

A modern exchange is a complicated system with stringent requirements on latency, throughput, and robustness. Before we start, let's ask the interviewer a few questions to clarify the requirements.

Candidate: Which securities are we going to trade? Stocks, options, or futures?
Interviewer: For simplicity, only stocks.

Candidate: Which types of order operations are supported: placing a new order, canceling an order, or replacing an order? Do we need to support limit order, market order, or conditional order?
Interviewer: We need to support the following: placing a new order and canceling an order. For the order type, we only need to consider the limit order.

Candidate: Does the system need to support after-hours trading?
Interviewer: No, we just need to support the normal trading hours.

Candidate: Could you describe the basic functions of the exchange? And the scale of the exchange, such as how many users, how many symbols, and how many orders?
Interviewer: A client can place new limit orders or cancel them, and receive matched trades in real-time. A client can view the real-time order book (the list of buy and sell orders). The exchange needs to support at least tens of thousands of users trading at the same time, and it needs to support at least 100 symbols. For the trading volume, we should support billions of orders per day. Also, the exchange is a regulated facility, so we need to make sure it runs risk checks.

Candidate: Could you please elaborate on risk checks?
Interviewer: Let's just do simple risk checks. For example, a user can only trade a maximum of 1 million shares of Apple stock in one day.

Candidate: I noticed you didn't mention user wallet management. Is it something we also need to consider?

Interviewer: Good catch! We need to make sure users have sufficient funds when they place orders. If an order is waiting in the order book to be filled, the funds required for the order need to be withheld to prevent overspending.

Non-functional requirements

After checking with the interviewer for the functional requirements, we should determine the non-functional requirements. In fact, requirements like "at least 100 symbols" and "tens of thousands of users" tell us that the interviewer wants us to design a small-to-medium scale exchange. On top of this, we should make sure the design can be extended to support more symbols and users. Many interviewers focus on extensibility as an area for follow-up questions.

Here is a list of non-functional requirements:

- **Availability.** At least 99.99%. Availability is crucial for exchanges. Downtime, even seconds, can harm reputation.
- **Fault tolerance.** Fault tolerance and a fast recovery mechanism are needed to limit the impact of a production incident.
- **Latency.** The round-trip latency should be at the millisecond level, with a particular focus on the 99th percentile latency. The round trip latency is measured from the moment a market order enters the exchange to the point where the market order returns as a filled execution. A persistently high 99th percentile latency causes a terrible user experience for a small number of users.
- **Security.** The exchange should have an account management system. For legal and compliance, the exchange performs a KYC (Know Your Client) check to verify a user's identity before a new account is opened. For public resources, such as web pages containing market data, we should prevent distributed denial-of-service (DDoS) [6] attacks.

Back-of-the-envelope estimation

Let's do some simple back-of-the-envelope calculations to understand the scale of the system:

- 100 symbols
- 1 billion orders per day
- NYSE Stock exchange is open Monday through Friday from 9:30 am to 4:00 pm Eastern Time. That's 6.5 hours in total.
- QPS: $\frac{1 \text{ billion}}{6.5 \times 3{,}600} = \sim 43{,}000$
- Peak QPS: $5 \times \text{QPS} = 215{,}000$. The trading volume is significantly higher when the market first opens in the morning and before it closes in the afternoon.

Step 2 - Propose High-Level Design and Get Buy-In

Before we dive into the high-level design, let's briefly discuss some basic concepts and terminology that are helpful for designing an exchange.

Business Knowledge 101

Broker

Most retail clients trade with an exchange via a broker. Some brokers whom you might be familiar with include Charles Schwab, Robinhood, E*Trade, Fidelity, etc. These brokers provide a friendly user interface for retail users to place trades and view market data.

Institutional client

Institutional clients trade in large volumes using specialized trading software. Different institutional clients operate with different requirements. For example, pension funds aim for a stable income. They trade infrequently, but when they do trade, the volume is large. They need features like order splitting to minimize the market impact [7] of their sizable orders. Some hedge funds specialize in market making and earn income via commission rebates. They need low latency trading abilities, so obviously they cannot simply view market data on a web page or a mobile app, as retail clients do.

Limit order

A limit order is a buy or sell order with a fixed price. It might not find a match immediately, or it might just be partially matched.

Market order

A market order doesn't specify a price. It is executed at the prevailing market price immediately. A market order sacrifices cost in order to guarantee execution. It is useful in certain fast-moving market conditions.

Market data levels

The US stock market has three tiers of price quotes: L1 (level 1), L2, and L3. L1 market data contains the best bid price, ask price, and quantities (Figure 13.2). Bid price refers to the highest price a buyer is willing to pay for a stock. Ask price refers to the lowest price a seller is willing to sell the stock.

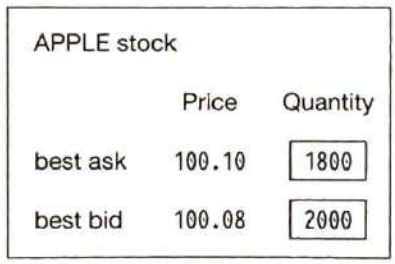


Figure 13.2: Level 1 data

L2 includes more price levels than L1 (Figure 13.3).

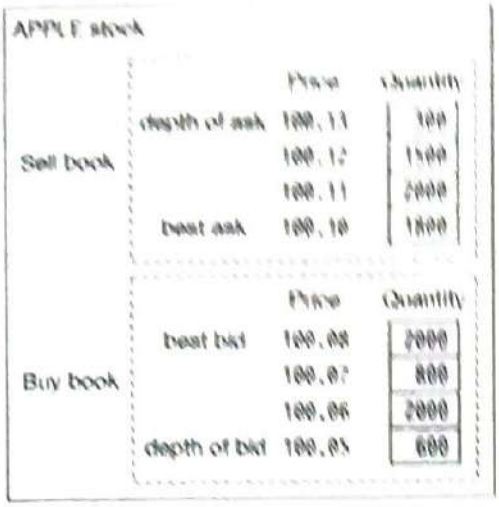


Figure 13.3: Level 2 data

shows price levels and the queued quantity at each price level (Figure 13.4).

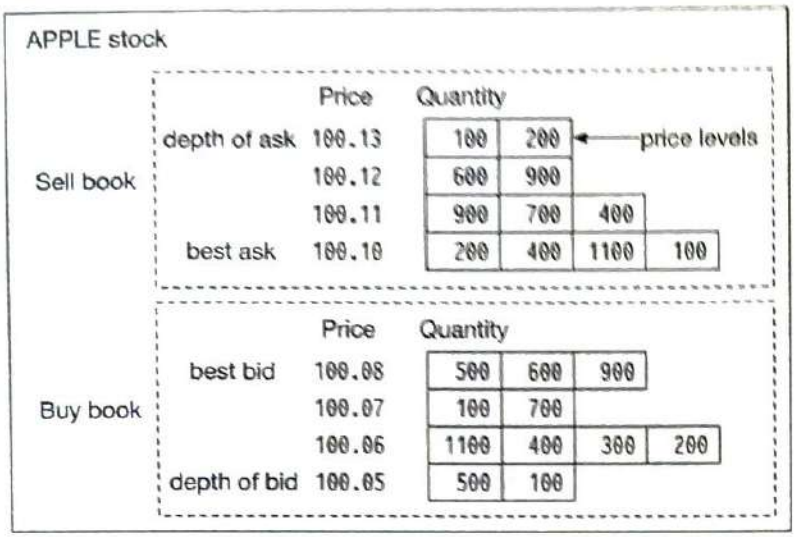


Figure 13.4: Level 3 data

Candlestick chart

A candlestick chart represents the stock price for a certain period of time. A typical candlestick looks like this (Figure 13.5). A candlestick shows the market's open, close, high, and low price for a time interval. The common time intervals are one-minute, five-minute, one-hour, one-day, one-week, and one-month.

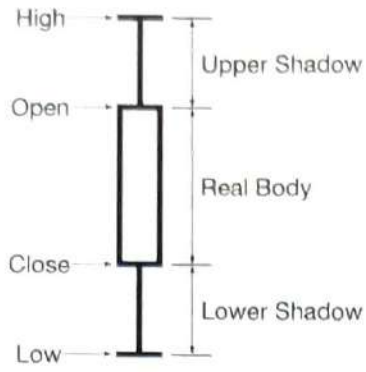


Figure 13.5: A single candlestick chart

FIX

FIX protocol [8], which stands for Financial Information exchange protocol, was created in 1991. It is a vendor-neutral communications protocol for exchanging securities transaction information. See below for an example of a securities transaction encoded in FIX [8].

```
8=FIX.4.2 | 9=176 | 35=8 | 49=PHLX | 56=PERS |
52=20071123-05:30:00.000 | 11=ATOMNOCCC9990900 | 20=3 | 150=E | 39=E
 | 55=MSFT | 167=CS | 54=1 | 38=15 | 40=2 | 44=15 | 58=PHLX EQUITY
TESTING | 59=0 | 47=C | 32=0 | 31=0 | 151=15 | 14=0 | 6=0 | 10=128 |
```

High-level design

Now that we have some basic understanding of the key concepts, let's take a look at the high-level design, as shown in Figure 13.6.

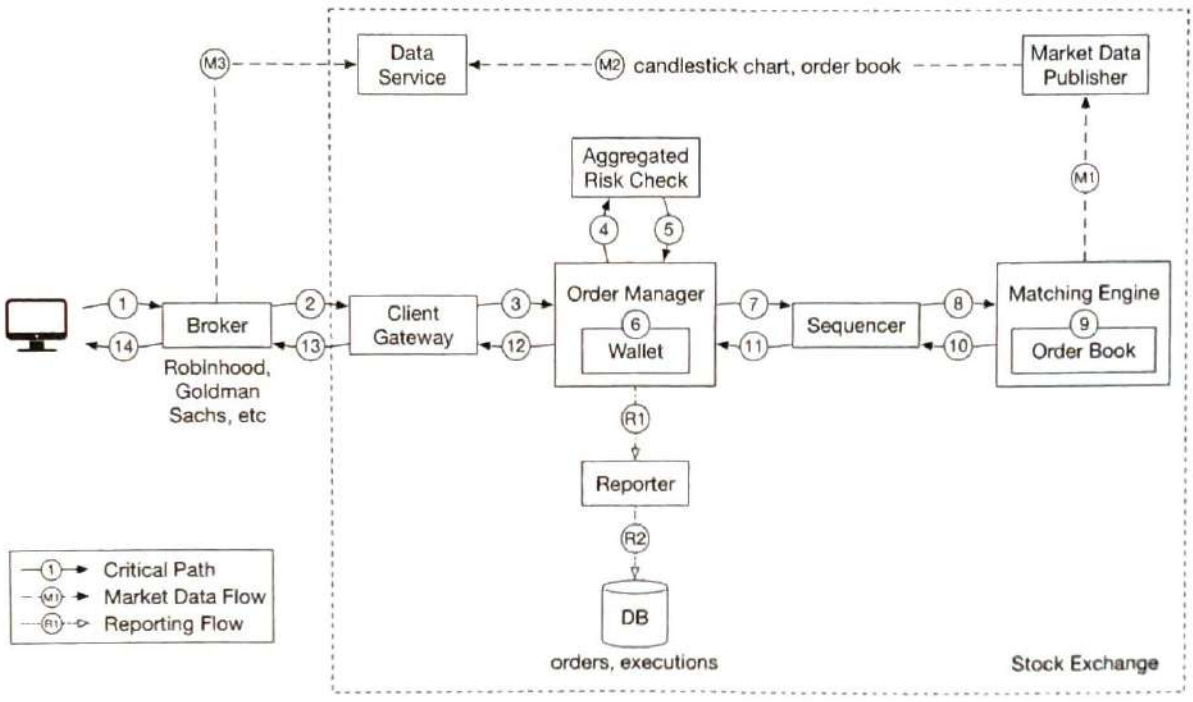


Figure 13.6: High-level design

Let's trace the life of an order through various components in the diagram to see how the pieces fit together.

First, we follow the order through the **trading flow**. This is the critical path with strict latency requirements. Everything has to happen fast in the flow:

Step 1: A client places an order via the broker's web or mobile app.

Step 2: The broker sends the order to the exchange.

Step 3: The order enters the exchange through the client gateway. The client gateway performs basic gatekeeping functions such as input validation, rate limiting, authentication, normalization, etc. The client gateway then forwards the order to the order manager.

Step 4 ~ 5: The order manager performs risk checks based on rules set by the risk manager.

Step 6: After passing risk checks, the order manager verifies there are sufficient funds in the wallet for the order.

Step 7 ~ 9: The order is sent to the matching engine. When a match is found, the matching engine emits two executions (also called fills), with one each for the buy and sell sides. To guarantee that matching results are deterministic when replayed, both orders and executions are sequenced in the sequencer (more on the sequencer later).

Step 10 ~ 14: The executions are returned to the client.

Next, we follow the **market data flow** and trace the order executions from the matching engine to the broker via the data service.

Step M1: The matching engine generates a stream of executions (fills) as matches are made. The stream is sent to the market data publisher.

Step M2: The market data publisher constructs the candlestick charts and the order books as market data from the stream of executions and orders. It then sends market data to the data service.

Step M3: The market data is saved to specialized storage for real-time analytics. Brokers connect to the data service to obtain timely market data. Brokers relay market data to their clients.

Lastly, we examine the **reporting flow**.

Step R1~R2 (reporting flow): The reporter collects all the necessary reporting fields (e.g. `client_id`, `price`, `quantity`, `order_type`, `filled_quantity`, `remaining_quantity`) from orders and executions, and writes the consolidated records to the database.

Note that the trading flow (steps 1 to 14) is on the critical path, while the market data flow and reporting flow are not. They have different latency requirements.

Now let's examine each of the three flows in more detail.

Trading flow

The trading flow is on the critical path of the exchange. Everything must happen fast. The heart of the trading flow is the matching engine. Let's go over that first.

Matching engine

The matching engine is also called the cross engine. Here are the primary responsibilities of the matching engine:

1. Maintain the order book for each symbol. An order book is a list of buy and sell orders for a symbol. We explain the construction of an order book in the Data models section later.
2. Match buy and sell orders. A match results in two executions (one from the buy side and the other from the sell side). The matching function must be fast and accurate.
3. Distribute the execution stream as market data.

A highly available matching engine implementation must be able to produce matches in a deterministic order. That is, given a known sequence of orders as an input, the matching engine must produce the same sequence of executions (fills) as an output when the sequence is replayed. This determinism is a foundation of high availability which we will discuss at length in the deep dive section.

Sequencer

The sequencer is the key component that makes the matching engine deterministic. It stamps every incoming order with a sequence ID before it is processed by the matching engine. It also stamps every pair of executions (fills) completed by the matching engine with sequence IDs. In other words, the sequencer has an inbound and an outbound instance, with each maintaining its own sequences. The sequence generated by each sequencer must be sequential numbers, so that any missing numbers can be easily detected. See Figure 13.7 for details.

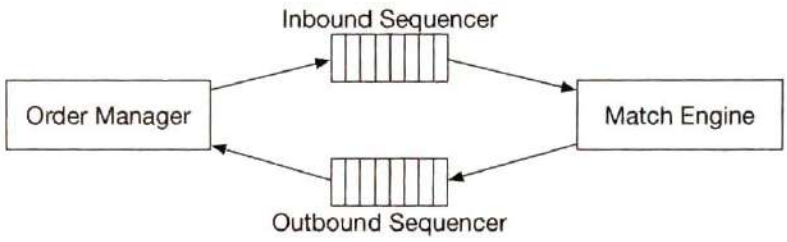


Figure 13.7: Inbound and outbound sequencers

The incoming orders and outgoing executions are stamped with sequence IDs for these reasons:

1. Timeliness and fairness
2. Fast recovery / replay
3. Exactly-once guarantee

The sequencer does not only generate sequence IDs. It also functions as a message queue. There is one to send messages (incoming orders) to the matching engine, and another one to send messages (executions) back to the order manager. It is also an event store for the orders and executions. It is similar to having two Kafka event streams connected to the matching engine, one for incoming orders and the other for outgoing executions. In fact, we could have used Kafka if its latency was lower and more predictable. We discuss how the sequencer is implemented in a low-latency exchange environment in the deep dive section.

Order manager

The order manager receives orders on one end and receives executions on the other. It manages the orders' states. Let's look at it closely.

The order manager receives inbound orders from the client gateway and performs the following:

- It sends the order for risk checks. Our requirements for risk checking are simple. For example, we verify that a user's trade volume is below $1M a day.
- It checks the order against the user's wallet and verifies that there are sufficient funds to cover the trade. The wallet was discussed at length in the "Digital Wallet" chapter on page 341. Refer to that chapter for an implementation that would work in the exchange.
- It sends the order to the sequencer where the order is stamped with a sequence ID. The sequenced order is then processed by the matching engine. There are many attributes in a new order, but there is no need to send all the attributes to the matching engine. To reduce the size of the message in data transmission, the order manager only sends the necessary attributes.

On the other end, the order manager receives executions from the matching engine via the sequencer. The order manager returns the executions for the filled orders to the brokers via the client gateway.

The order manager should be fast, efficient, and accurate. It maintains the current states for the orders. In fact, the challenge of managing the various state transitions is the major source of complexity for the order manager. There can be tens of thousands of cases involved in a real exchange system. Event sourcing [9] is perfect for the design of an order manager. We discuss an event sourcing design in the deep dive section.

Client gateway

The client gateway is the gatekeeper for the exchange. It receives orders placed by clients and routes them to the order manager. The gateway provides the following functions as shown in Figure 13.8.

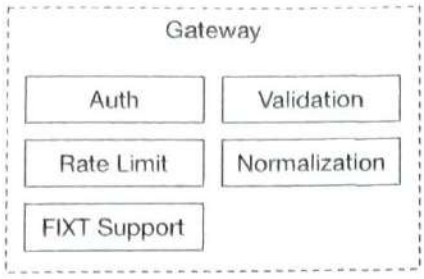


Figure 13.8: Client gateway components

The client gateway is on the critical path and is latency-sensitive. It should stay lightweight. It passes orders to the correct destinations as quickly as possible. The functions above, while critical, must be completed as quickly as possible. It is a design trade-off to decide what functionality to put in the client gateway, and what to leave out. As a general guideline, we should leave complicated functions to the matching engine and risk check.

There are different types of client gateways for retail and institutional clients. The main considerations are latency, transaction volume, and security requirements. For instance, institutions like the market makers provide a large portion of liquidity for the exchange. They require very low latency. Figure 13.9 shows different client gateway connections to an exchange. An extreme example is the colocation (colo) engine. It is the trading engine software running on some servers rented by the broker in the exchange's data center. The latency is literally the time it takes for light to travel from the colocated server to the exchange server [10].

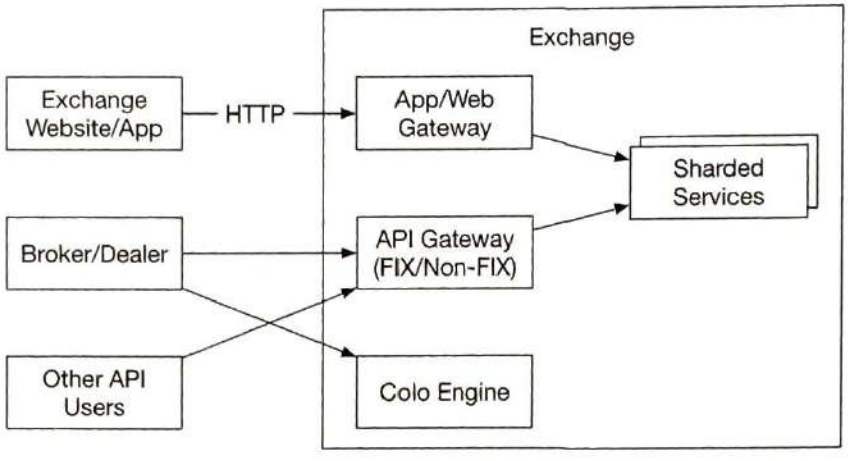


Figure 13.9: Client gateway

Market data flow

The market data publisher (MDP) receives executions from the matching engine and builds the order books and candlestick charts from the stream of executions. The order books and candlestick charts, which we discuss in the Data Models section later, are collectively called market data. The market data is sent to the data service where they are made available to subscribers. Figure 13.10 shows an implementation of MDP and how it fits with the other components in the market data flow.

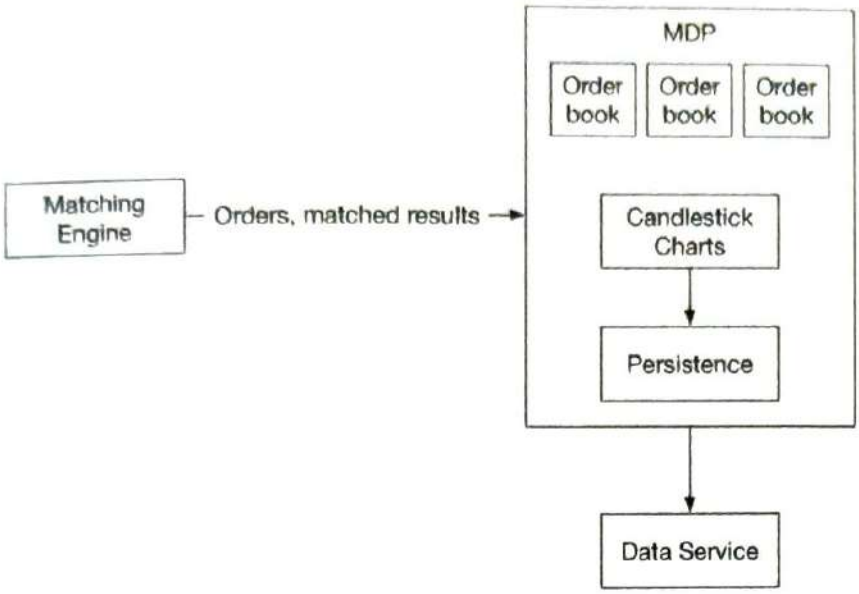


Figure 13.10: Market Data Publisher

Reporting flow

One essential part of the exchange is reporting. The reporter is not on the trading critical path, but it is a critical part of the system. It provides trading history, tax reporting, compliance reporting, settlements, etc. Efficiency and latency are critical for the trading flow, but the reporter is less sensitive to latency. Accuracy and compliance are key factors for the reporter.

It is common practice to piece attributes together from both incoming orders and outgoing executions. An incoming new order contains order details, and outgoing execution usually only contains order ID, price, quantity, and execution status. The reporter merges the attributes from both sources for the reports. Figure 13.11 shows how the components in the reporting flow fit together.

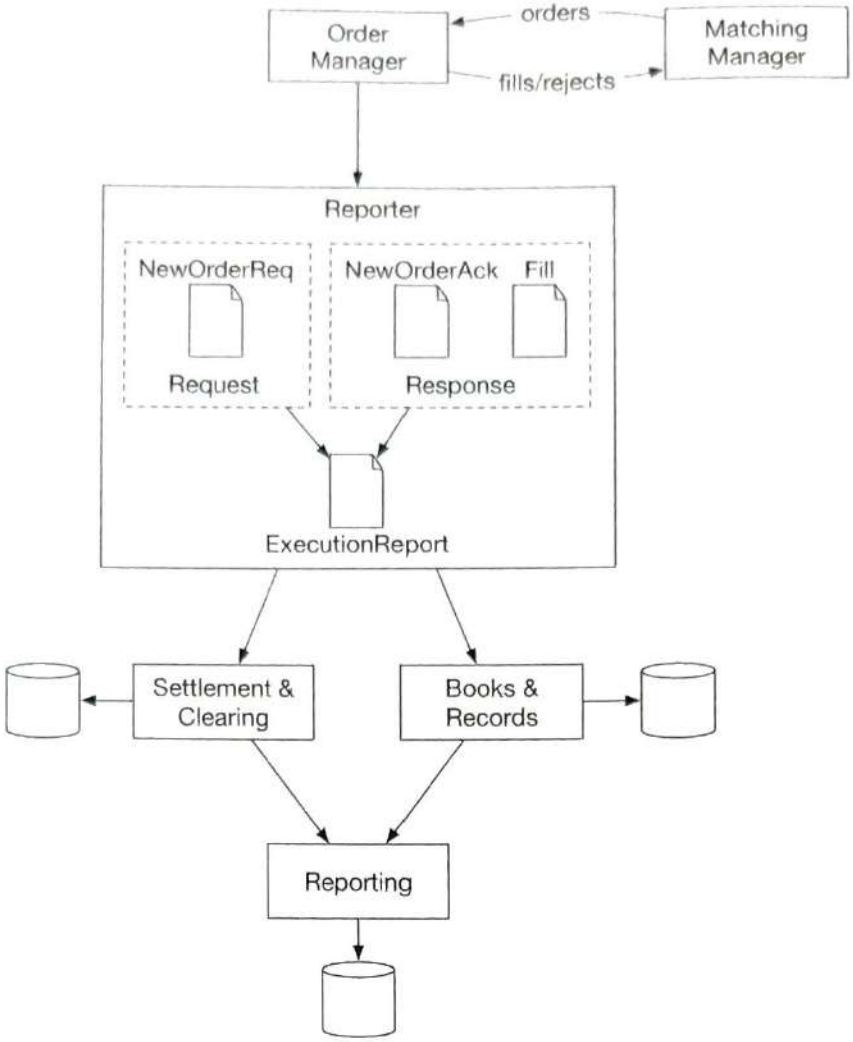


Figure 13.11: Reporter

A sharp reader might notice that the section order of "Step 2 - Propose High-Level Design and Get Buy-In" looks a little different than other chapters. In this chapter, the API design and data models sections come after the high-level design. The sections are arranged this way because these other sections require some concepts that were introduced in the high-level design.

API Design

Now that we understand the high-level design, let's take a look at the API design.

Clients interact with the stock exchange via the brokers to place orders, view executions, view market data, download historical data for analysis, etc. We use the RESTful conventions for the API below to specify the interface between the brokers and the client gateway. Refer to the "Data models" section on page 393 for the resources mentioned below.

Note that the RESTful API might not satisfy the latency requirements of institutional clients like hedge funds. The specialized software built for these institutions likely uses a different protocol, but no matter what it is, the basic functionality mentioned below

needs to be supported.

Order

`POST /v1/order`

This endpoint places an order. It requires authentication.

Parameters

`symbol`: the stock symbol. String
`side`: buy or sell. String
`price`: the price of the limit order. Long
`orderType`: limit or market (note we only support limit orders in our design). String
`quantity`: the quantity of the order. Long

Response

Body:

`id`: the ID of the order. Long
`creationTime`: the system creation time of the order. Long
`filledQuantity`: the quantity that has been successfully executed. Long
`remainingQuantity`: the quantity still to be executed. Long
`status`: `new/canceled/filled`. String
rest of the attributes are the same as the input parameters

Code:

200: successful
40x: parameter error/access denied/unauthorized
500: server error

Execution

`GET /v1/execution?symbol={:symbol}&orderId={:orderId}&startTime={:startTime}&endTime={:endTime}`

This endpoint queries execution info. It requires authentication.

Parameters

`symbol`: the stock symbol. String
`orderId`: the ID of the order. Optional. String
`startTime`: query start time in epoch [11]. Long
`endTime`: query end time in epoch. Long

Response

Body:

executions: array with each execution in scope (see attributes below). Array
id: the ID of the execution. Long
orderId: the ID of the order. Long
symbol: the stock symbol. String
side: buy or sell. String
price: the price of the execution. Long
orderType: limit or market. String
quantity: the filled quantity. Long

Code:

200: successful
40x: parameter error/not found/access denied/unauthorized
500: server error

Order book

`GET /v1/marketdata/orderBook/L2?symbol={:symbol}&depth={:depth}`

This endpoint queries L2 order book information for a symbol with designated depth.

Parameters

symbol: the stock symbol. String
depth: order book depth per side. Int
startTime: query start time in epoch. Long
endTime: query end time in epoch. Long

Response

Body:

bids: array with price and size. Array
asks: array with price and size.Array

Code:

200: successful
40x: parameter error/not found/access denied/unauthorized
500: server error

Historical prices (candlestick charts)

`GET /v1/marketdata/candles?symbol={:symbol}&resolution={:resolution}&startTime {:startTime}&endTime={:endTime}`

This endpoint queries candlestick chart data (see candlestick chart in data models section) for a symbol given a time range and resolution.

Parameters

`symbol`: the stock symbol. String
`resolution`: window length of the candlestick chart in seconds. Long
`startTime`: start time of the window in epoch. Long
`endTime`: end time of the window in epoch. Long

Response

Body:

candles: array with each candlestick data (attributes listed below). Array
`open`: open price of each candlestick. Double
`close`: close price of each candlestick. Double
`high`: high price of each candlestick. Double
`low`: low price of each candlestick. Double

Code:

200: successful
40x: parameter error/not found/access denied/unauthorized
500: server error

Data models

There are three main types of data in the stock exchange. Let's explore them one by one.

- Product, order, and execution
- Order book
- Candlestick chart

Product, order, execution

A product describes the attributes of a traded symbol, like product type, trading symbol, UI display symbol, settlement currency, lot size, tick size, etc. This data doesn't change frequently. It is primarily used for UI display. The data can be stored in any database and is highly cacheable.

An order represents the inbound instruction for a buy or sell order. An execution represents the outbound matched result. An execution is also called a fill. Not every order has an execution. The output of the matching engine contains two executions, representing the buy and sell sides of a matched order.

See Figure 13.12 for the logical model diagram that shows the relationships between the three entities. Note it is not a database schema.

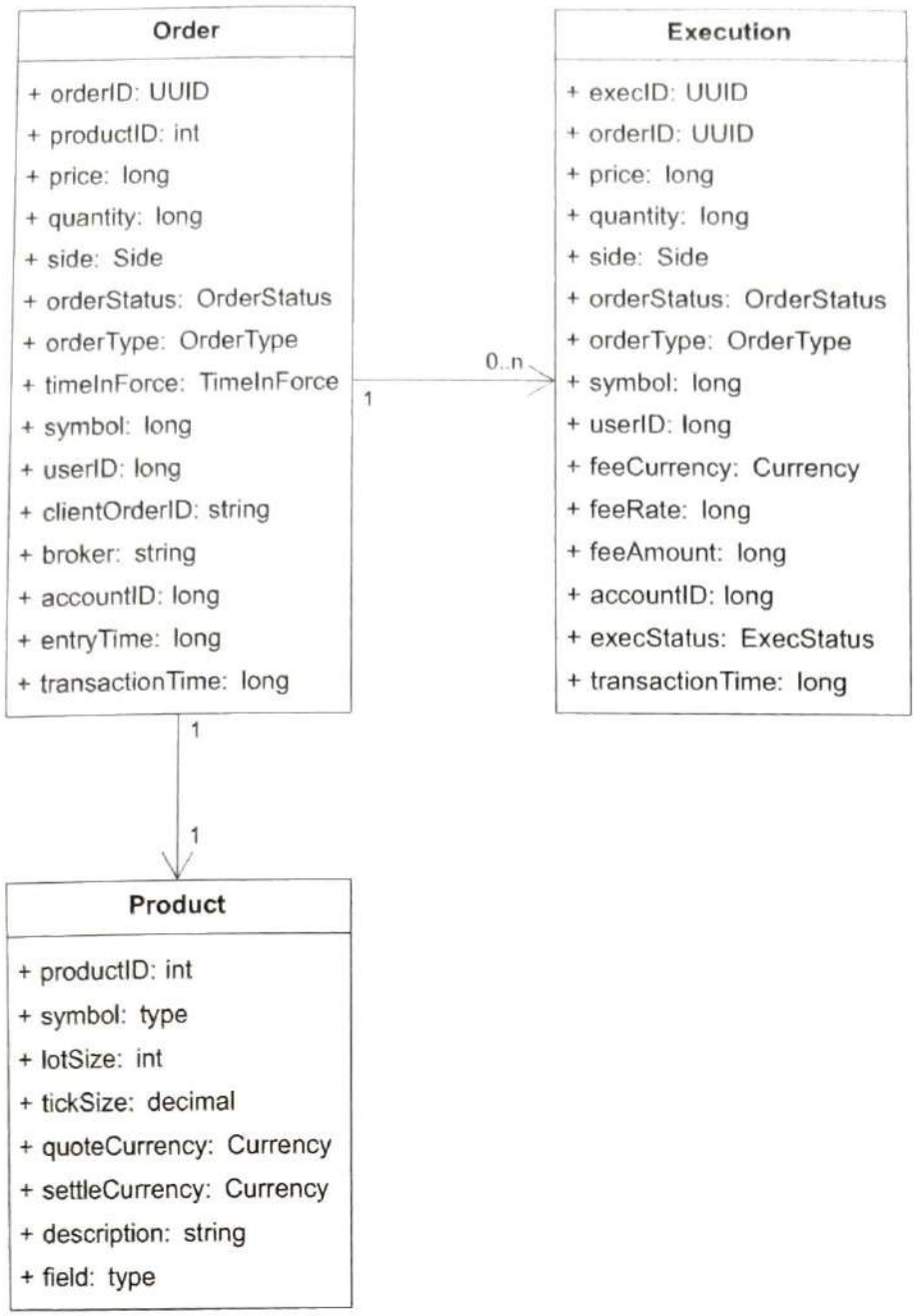


Figure 13.12: Product, order, execution

Orders and executions are the most important data in the exchange. We encounter them in all three flows mentioned in the high-level design, in slightly different forms.

- In the critical trading path, orders and executions are not stored in a database. To achieve high performance, this path executes trades in memory and leverages hard disk or shared memory to persist and share orders and executions. Specifically, orders and executions are stored in the sequencer for fast recovery, and data is archived after the market closes. We discuss an efficient implementation of the sequencer in the deep dive section.
- The reporter writes orders and executions to the database for reporting use cases like reconciliation and tax reporting.

- Executions are forwarded to the market data processor to reconstruct the order book and candlestick chart data. We discuss these data types next.

Order book

An order book is a list of buy and sell orders for a specific security or financial instrument, organized by price level [12] [13]. It is a key data structure in the matching engine for fast order matching. An efficient data structure for an order book must satisfy these requirements:

- Constant lookup time. Operation includes: getting volume at a price level or between price levels.
- Fast add/cancel/execute operations, preferably $O(1)$ time complexity. Operations include: placing a new order, canceling an order, and matching an order.
- Fast update. Operation: replacing an order.
- Query best bid/ask.
- Iterate through price levels.

Let's walk through an example order execution against an order book, as illustrated in Figure 13.13.

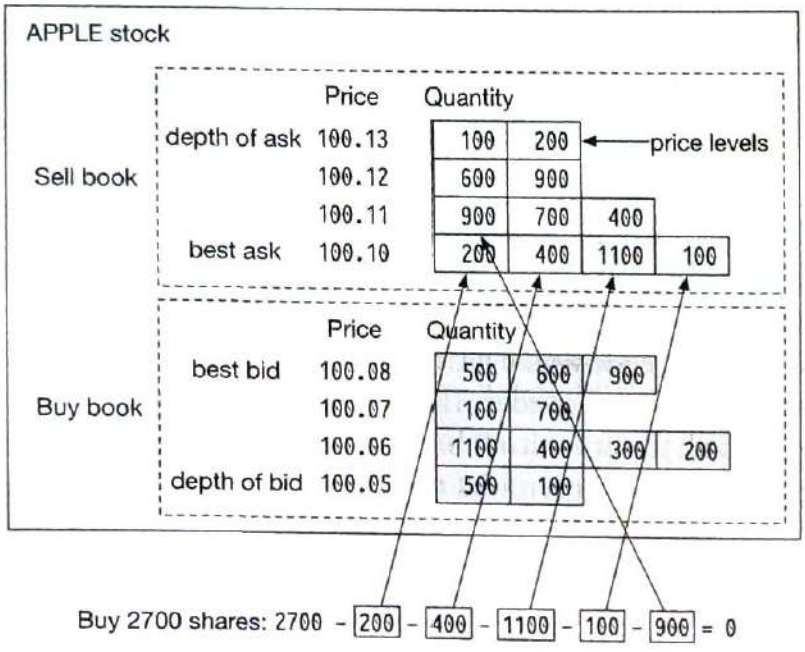


Figure 13.13: Limit order book illustrated

In the example above, there is a large market buy order for 2,700 shares of Apple. The buy order matches all the sell orders in the best ask queue and the first sell order in the 100.11 price queue. After fulfilling this large order, the bid/ask spread widens, and the price increases by one level (best ask is 100.11 now).

The following code snippet shows an implementation of the order book.

```
class PriceLevel{
  private Price limitPrice;
  private long totalVolume;
  private List<Order> orders;
}
class Book<Side> {
  private Side side;
  private Map<Price, PriceLevel> limitMap;
}
class OrderBook {
  private Book<Buy> buyBook;
  private Book<Sell> sellBook;
  private PriceLevel bestBid;
  private PriceLevel bestOffer;
  private Map<OrderID, Order> orderMap;
}
```

Does the code meet all the design requirements stated above? For example, when adding/canceling a limit order, is the time complexity $O(1)$? The answer is no since we are using a plain list here (`private List<Order> orders`). To have a more efficient order book, change the data structure of "orders" to a doubly-linked list so that the deletion type of operation (cancel and match) is also $O(1)$. Let's review how we achieve $O(1)$ time complexity for these operations:

1. Placing a new order means adding a new `Order` to the tail of the `PriceLevel`. This is $O(1)$ time complexity for a doubly-linked list.
2. Matching an order means deleting an `Order` from the head of the PriceLevel. This is $O(1)$ time complexity for a doubly-linked list.
3. Canceling an order means deleting an `Order` from the OrderBook. We leverage the helper data structure `Map<OrderID, Order> orderMap` in the `OrderBook` to find the `Order` to cancel in $O(1)$ time. Once the order is found, if the "orders" list was a singly-linked list, the code would have to traverse the entire list to locate the previous pointer in order to delete the order. That would have taken $O(n)$ time. Since the list is now doubly-linked, the order itself has a pointer to the previous order, which allows the code to delete the order without traversing the entire order list.

Figure 13.14 explains how these three operations work.

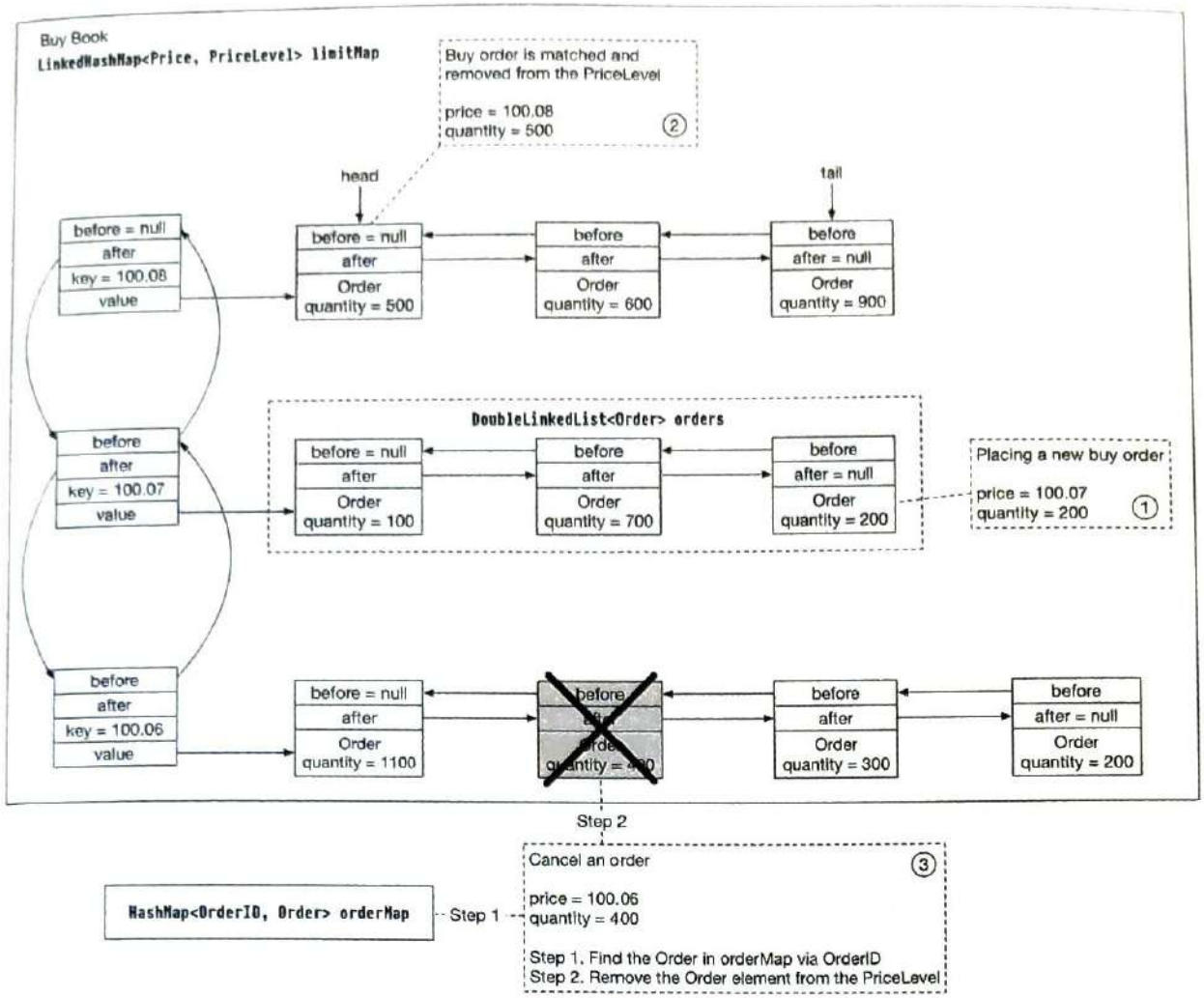


Figure 13.14: Place, match, and cancel an order in $O(1)$

See the reference material for more details [14].

It is worth noting that the order book data structure is also heavily used in the market data processor to reconstruct the L1, L2, and L3 data from the streams of executions generated by the matching engine.

Candlestick chart

Candlestick chart is another key data structure (alongside order book) in the market data processor to produce market data.

We model this with a `Candlestick` class and a `CandlestickChart` class. When the interval for the candlestick has elapsed, a new `Candlestick` class is instantiated for the next interval and added to the linked list in the `CandleStickChart` instance.

```
class Candlestick {
  private long openPrice;
  private long closePrice;
  private long highPrice;
  private long lowPrice;
  private long volume;
  private long timestamp;
```

```
    private int interval;
}
class CandlestickChart {
    private LinkedList<Candlestick> sticks;
}
```

Tracking price history in candlestick charts for many symbols at many time intervals consumes a lot of memory. How can we optimize it? Here are two ways:

1. Use pre-allocated ring buffers to hold sticks to reduce the number of new object allocations.
2. Limit the number of sticks in the memory and persist the rest to disk.

We will examine the optimizations in the "Market data publisher" section in deep dive on page 409.

The market data is usually persisted in an in-memory columnar database (for example, KDB [15]) for real-time analytics. After the market is closed, data is persisted in a historical database.

Step 3 - Design Deep Dive

Now that we understand how an exchange works at a high level, let's investigate how a modern exchange has evolved to become what it is today. What does a modern exchange look like? The answer might surprise a lot of readers. Some large exchanges run almost everything on a single gigantic server. While it might sound extreme, we can learn many good lessons from it.

Let's dive in.

Performance

As discussed in the non-functional requirements, latency is very important for an exchange. Not only does the average latency need to be low, but the overall latency must also be stable. A good measure for the level of stability is the 99th percentile latency.

Latency can be broken down into its components as shown in the formula below:

$$\text{Latency} = \sum \text{executionTimeAlongCriticalPath}$$

There are two ways to reduce latency:

1. Decrease the number of tasks on the critical path.
2. Shorten the time spent on each task:
    a. By reducing or eliminating network and disk usage
    b. By reducing execution time for each task

Let's review the first point. As shown in the high-level design, the critical trading path includes the following:

$$gateway \rightarrow order\ manager \rightarrow sequencer \rightarrow matching\ engine$$

The critical path only contains the necessary components, even logging is removed from the critical path to achieve low latency.

Now let's look at the second point. In the high-level design, the components on the critical path run on individual servers connected over the network. The round trip network latency is about 500 microseconds. When there are multiple components all communicating over the network on the critical path, the total network latency adds up to single-digit milliseconds. In addition, the sequencer is an event store that persists events to disk. Even assuming an efficient design that leverages the performance advantage of sequential writes, the latency of disk access still measures in tens of milliseconds. To learn more about network and disk access latency, see "Latency Numbers Every Programmer Should Know" [16].

Accounting for both network and disk access latency, the total end-to-end latency adds up to tens of milliseconds. While this number was respectable in the early days of the exchange, it is no longer sufficient as exchanges compete for ultra-low latency.

To stay ahead of the competition, exchanges over time evolve their design to reduce the end-to-end latency on the critical path to tens of microseconds, primarily by exploring options to reduce or eliminate network and disk access latency. A time-tested design eliminates the network hops by putting everything on the same server. When all components are on the same server, they can communicate via mmap [17] as an event store (more on this later).

Figure 13.15 shows a low-latency design with all the components on a single server:

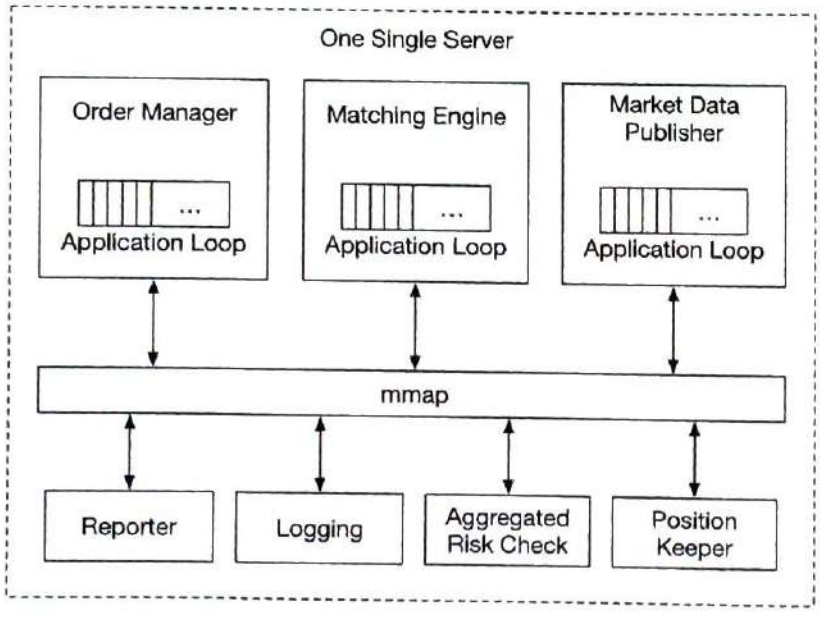


Figure 13.15: A low latency single server exchange design

There are a few interesting design decisions that are worth a closer look at.

Let's first focus on the application loops in the diagram above. An application loop is an interesting concept. It keeps polling for tasks to execute in a while loop and is the primary task execution mechanism. To meet the strict latency budget, only the most mission-critical tasks should be processed by the application loop. Its goal is to reduce the execution time for each component and to guarantee a highly predictable execution time (i.e., a low 99th percentile latency). Each box in the diagram represents a component. A component is a process on the server. To maximize CPU efficiency, each application loop (think of it as the main processing loop) is single-threaded, and the thread is pinned to a fixed CPU core. Using the order manager as an example, it looks like the following diagram (Figure 13.16).

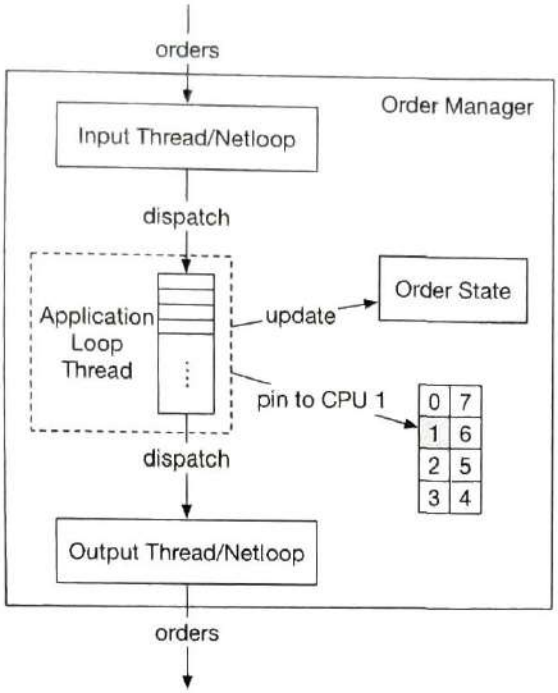


Figure 13.16: Application loop thread in Order Manager

In this diagram, the application loop for the order manager is pinned to CPU 1. The benefits of pinning the application loop to the CPU are substantial:

1. No context switch [18]. CPU 1 is fully allocated to the order manager's application loop.
2. No locks and therefore no lock contention, since there is only one thread that updates states.

Both of these contribute to a low 99th percentile latency.

The tradeoff of CPU pinning is that it makes coding more complicated. Engineers need to carefully analyze the time each task takes to keep it from occupying the application loop thread for too long, as it can potentially block subsequent tasks.

Next, let's focus our attention on the long rectangle labeled "mmap" at the center of

Figure 13.15. "mmap" refers to a POSIX-compliant UNIX system call named `mmap(2)` that maps a file into the memory of a process.

`mmap(2)` provides a mechanism for high-performance sharing of memory between processes. The performance advantage is compounded when the backing file is in `/dev/shm`. `/dev/shm` is a memory-backed file system. When `mmap(2)` is done over a file in `/dev/shm`, the access to the shared memory does not result in any disk access at all.

Modern exchanges take advantage of this to eliminate as much disk access from the critical path as possible. `mmap(2)` is used in the server to implement a message bus over which the components on the critical path communicate. The communication pathway has no network or disk access, and sending a message on this mmap message bus takes sub-microsecond. By leveraging mmap to build an event store, coupled with the event sourcing design paradigm which we will discuss next, modern exchanges can build low-latency microservices inside a server.

Event sourcing

We discussed event sourcing in the "Digital Wallet" chapter on page 341. Please refer to that chapter for an in-depth review of event sourcing.

The concept of event sourcing is not hard to understand. In a traditional application, states are persisted in a database. When something goes wrong, it is hard to trace the source of the issue. The database only keeps the current states, and there are no records of the events that have led to the current states.

In event sourcing, instead of storing the current states, it keeps an immutable log of all state-changing events. These events are the golden source of truth. See Figure 13.17 for a comparison.

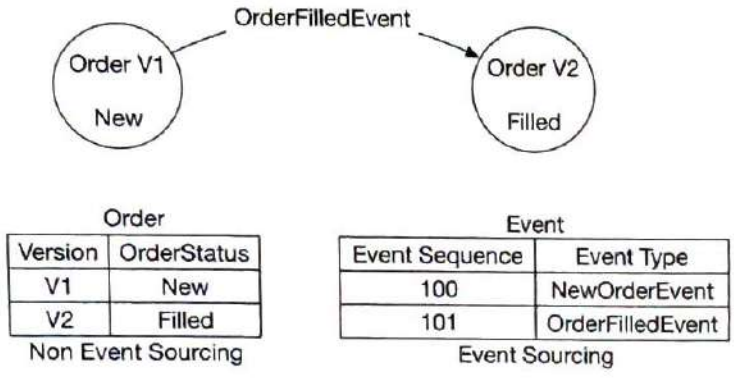


Figure 13.17: Non-event sourcing vs event sourcing

On the left is a classic database schema. It keeps track of the order status for an order, but it does not contain any information about how an order arrives at the current state. On the right is the event sourcing counterpart. It tracks all the events that change the order states, and it can recover order states by replaying all the events in sequence.

Figure 13.18 shows an event sourcing design using the mmap event store as a message bus. This looks very much like the Pub-Sub model in Kafka. In fact, if there is no strict

latency requirement, Kafka could be used.

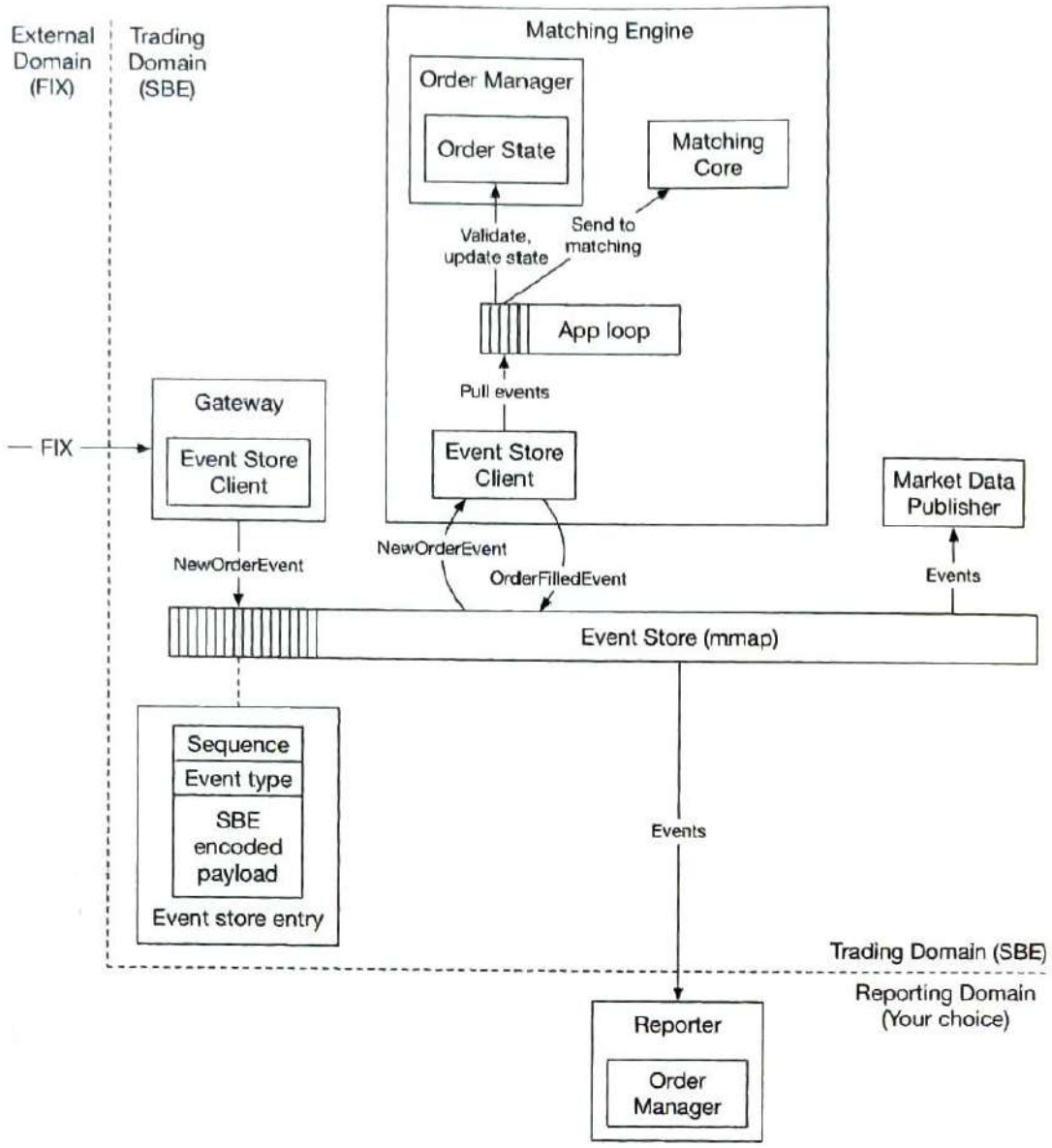


Figure 13.18: An event sourcing design

In the diagram, the external domain communicates with the trading domain using FIX that we introduced in the Business Knowledge 101 section on page 382.

- The gateway transforms FIX to "FIX over Simple Binary Encoding" (SBE) for fast and compact encoding and sends each order as a `NewOrderEvent` via the Event Store Client in a pre-defined format (see event store entry in the diagram).
- The order manager (embedded in the matching engine) receives the `NewOrderEvent` from the event store, validates it, and adds it to its internal order states. The order is then sent to the matching core.
- If the order gets matched, an `OrderFilledEvent` is generated and sent to the event store.

- Other components such as the market data processor and the reporter subscribe to the event store and process those events accordingly.

This design follows the high-level design closely, but there are some adjustments to make it work more efficiently in the event sourcing paradigm.

The first difference is the order manager. The order manager becomes a reusable library that is embedded in different components. It makes sense for this design because the states of the orders are important for multiple components. Having a centralized order manager for other components to update or query the order states would hurt latency, especially if those components are not on the critical trading path, as is the case for the reporter in the diagram. Although each component maintains the order states by itself, with event sourcing the states are guaranteed to be identical and replayable.

Another key difference is that the sequencer is nowhere to be seen. What happened to it?

With the event sourcing design, we have one single event store for all messages. Note that the event store entry contains a `sequence` field. This field is injected by the sequencer.

There is only one sequencer for each event store. It is a bad practice to have multiple sequencers, as they will fight for the right to write to the event store. In a busy system like an exchange, a lot of time would be wasted on lock contention. Therefore, the sequencer is a single writer which sequences the events before sending them to the event store. Unlike the sequencer in the high-level design which also functions as a message store, the sequencer here only does one simple thing and is super fast. Figure 13.19 shows a design for the sequencer in a memory-map (mmap) environment.

The sequencer pulls events from the ring buffer that is local to each component. For each event, it stamps a sequence ID on the event and sends it to the event store. We can have backup sequencers for high availability in case the primary sequencer goes down.

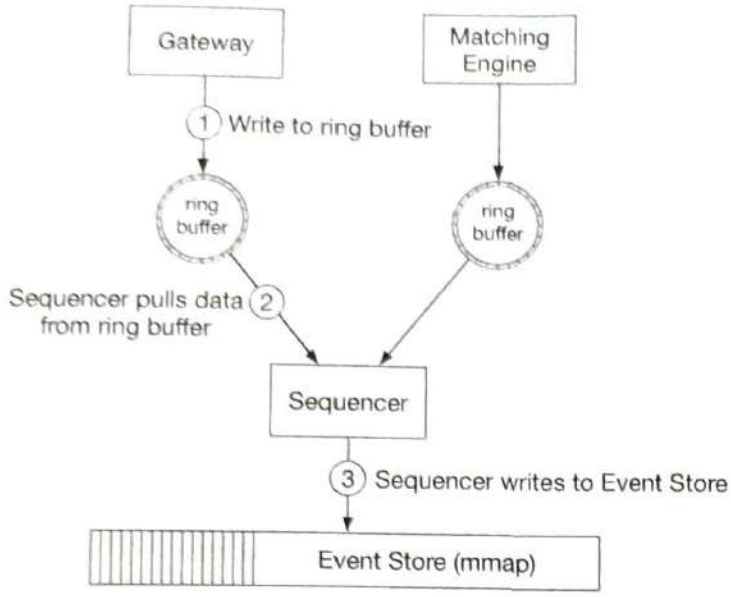


Figure 13.19: Sample design of Sequencer

High availability

For high availability, our design aims for 4 nines (99.99%). This means the exchange can only have 8.64 seconds of downtime per day. It requires almost immediate recovery if a service goes down.

To achieve high availability, consider the following:

- First, identify single-point-of-failures in the exchange architecture. For example, the failure of the matching engine could be a disaster for the exchange. Therefore, we set up redundant instances alongside the primary instance.
- Second, detection of failure and the decision to failover to the backup instance should be fast.

For stateless services such as the client gateway, they could easily be horizontally scaled by adding more servers. For stateful components, such as the order manager and matching engine, we need to be able to copy state data across replicas.

Figure 13.20 shows an example of how to copy data. The hot matching engine works as the primary instance, and the warm engine receives and processes the exact same events but does not send any event out onto the event store. When the primary goes down, the warm instance can immediately take over as the primary and send out events. When the warm secondary instance goes down, upon restart, it can always recover all the states from the event store. Event sourcing is a great fit for the exchange architecture. The inherent determinism makes state recovery easy and accurate.

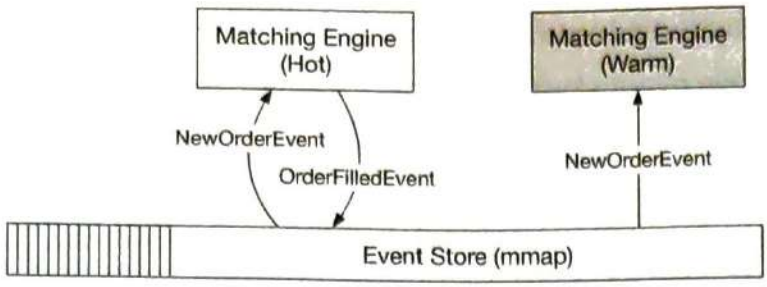


Figure 13.20: Hot-warm matching engine

We need to design a mechanism to detect potential problems in the primary. Besides normal monitoring of hardware and processes, we can also send heartbeats from the matching engine. If a heartbeat is not received in time, the matching engine might be experiencing problems.

The problem with this hot-warm design is that it only works within the boundary of a single server. To achieve high availability, we have to extend this concept across multiple machines or even across data centers. In this setting, an entire server is either hot or warm, and the entire event store is replicated from the hot server to all warm replicas. Replicating the entire event store across machines takes time. We could use reliable UDP [19] to efficiently broadcast the event messages to all warm servers. Refer to the design of Aeron [20] for an example.

In the next section, we discuss an improvement to the hot-warm design to achieve high availability.

Fault tolerance

The hot-warm design above is relatively simple. It works reasonably well, but what happens if the warm instances go down as well? This is a low probability but catastrophic event, so we should prepare for it.

This is a problem large tech companies face. They tackle it by replicating core data to data centers in multiple cities. It mitigates the risk of a natural disaster such as an earthquake or a large-scale power outage. To make the system fault-tolerant, we have to answer many questions:

1. If the primary instance goes down, how and when do we decide to failover to the backup instance?
2. How do we choose the leader among backup instances?
3. What is the recovery time needed (RTO - Recovery Time Objective)?
4. What functionalities need to be recovered (RPO - Recovery Point Objective)? Can our system operate under degraded conditions?

Let's answer these questions one by one.

First, we have to understand what "down" really means. This is not as straightforward as it seems. Consider these situations.

1. The system might send out false alarms, which cause unnecessary failovers.
2. Bugs in the code might cause the primary instance to go down. The same bug could bring down the backup instance after the failover. When all backup instances are knocked out by the bug, the system is no longer available.

These are tough problems to solve. Here are some suggestions. When we first release a new system, we might need to perform failovers manually. Only when we gather enough signals and operational experience and gain more confidence in the system do we automate the failure detection process. Chaos engineering [21] is a good practice to surface edge cases and gain operational experience faster.

Once the decision to failover is correctly made, how do we decide which server takes over? Fortunately, this is a well-understood problem. There are many battle-tested leader-election algorithms. We use Raft [22] as an example.

Figure 13.21 shows a Raft cluster with 5 servers with their own event stores. The current leader sends data to all the other instances (followers). The minimum number of votes required to perform an operation in Raft is $\frac{n}{2} + 1$, where n is the number of members in the cluster. In this example, the minimum is $\frac{5}{2} + 1 = 3$.

The following diagram (Figure 13.21) shows the followers receiving new events from the leader over RPC. The events are saved to the follower's own mmap event store.

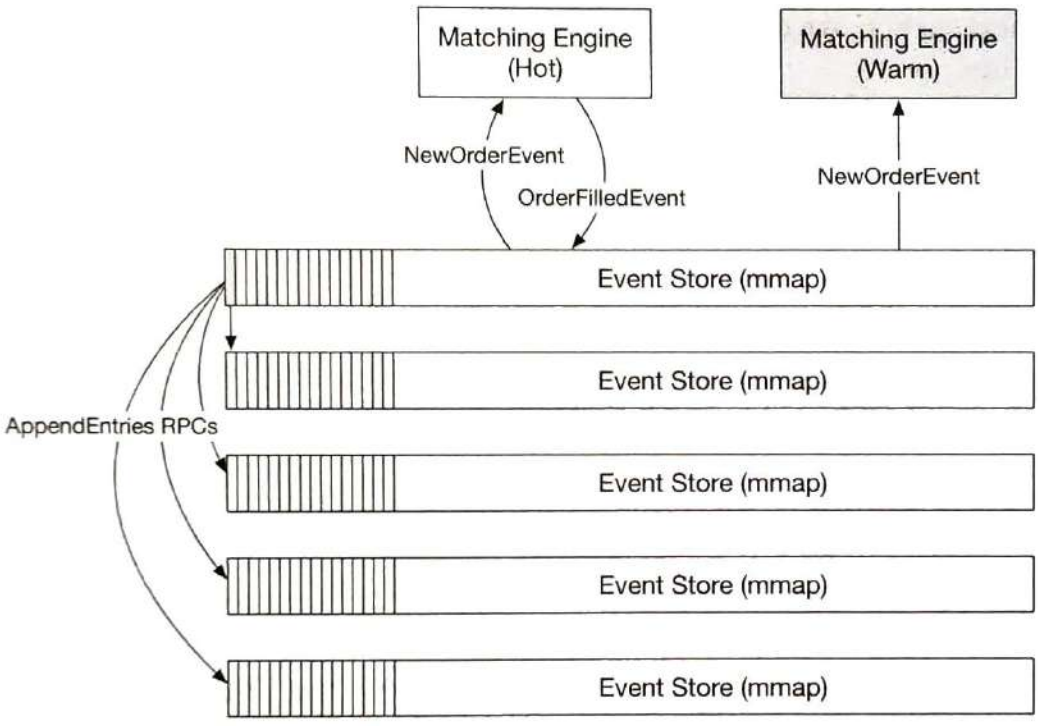


Figure 13.21: Event replication in Raft cluster

Let's briefly examine the leader election process. The leader sends heartbeat messages (`AppendEnties` with no content as shown in Figure 13.21) to its followers. If a follower has not received heartbeat messages for a period of time, it triggers an election timeout that initiates a new election. The first follower that reaches election timeout becomes a

candidate, and it asks the rest of the followers to vote (RequestVote). If the first follower receives a majority of votes, it becomes the new leader. If the first follower has a lower term value than the new node, it cannot be the leader. If multiple followers become candidates at the same time, it is called a "split vote". In this case, the election times out, and a new election is initiated. See Figure 13.22 for the explanation of "term". Time is divided into arbitrary intervals in Raft to represent normal operation and election.

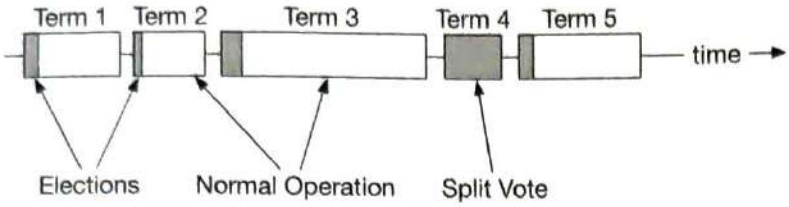


Figure 13.22: Raft terms (Source: [23])

Next, let's take a look at recovery time. Recovery Time Objective (RTO) refers to the amount of time an application can be down without causing significant damage to the business. For a stock exchange, we need to achieve a second-level RTO, which definitely requires automatic failover of services. To do this, we categorize services based on priority and define a degradation strategy to maintain a minimum service level.

Finally, we need to figure out the tolerance for data loss. Recovery Point Objective (RPO) refers to the amount of data that can be lost before significant harm is done to the business, i.e. the loss tolerance. In practice, this means backing up data frequently. For a stock exchange, data loss is not acceptable, so RPO is near zero. With Raft, we have many copies of the data. it guarantees that state consensus is achieved among cluster nodes. If the current leader crashes, the new leader should be able to function immediately.

Matching algorithms

Let's take a slight detour and dive into the matching algorithms. The pseudo-code below explains how matching works at a high level.

```
Context handleOrder(OrderBook orderBook, OrderEvent orderEvent) {
  if (orderEvent.getSequenceId() != nextSequence) {
    return Error(OUT_OF_ORDER, nextSequence);
  }

  if (!validateOrder(symbol, price, quantity)) {
    return ERROR(INVALID_ORDER, orderEvent);
  }

  Order order = createOrderFromEvent(orderEvent);
  switch (msgType):
    case NEW:
      return handleNew(orderBook, order);
    case CANCEL:
      return handleCancel(orderBook, order);
    default:
      return ERROR(INVALID_MSG_TYPE, msgType);
```

```
  }

  Context handleNew(OrderBook orderBook, Order order) {
    if (BUY.equals(order.side)) {
      return match(orderBook.sellBook, order);
    } else {
      return match(orderBook.buyBook, order);
    }
  }

  Context handleCancel(OrderBook orderBook, Order order) {
    if (!orderBook.orderMap.contains(order.orderId)) {
      return ERROR(CANNOT_CANCEL_ALREADY_MATCHED, order);
    }
    removeOrder(order);
    setOrderStatus(order, CANCELED);
    return SUCCESS(CANCEL_SUCCESS, order);
  }

  Context match(OrderBook book, Order order) {
    Quantity leavesQuantity = order.quantity - order.matchedQuantity;
    Iterator<Order> limitIter = book.limitMap.get(order.price).orders;
    while (limitIter.hasNext() && leavesQuantity > 0) {
      Quantity matched = min(limitIter.next.quantity, order.quantity);
      order.matchedQuantity += matched;
      leavesQuantity = order.quantity - order.matchedQuantity;
      remove(limitIter.next);
      generateMatchedFill();
    }
    return SUCCESS(MATCH_SUCCESS, order);
  }
```

The pseudocode uses the FIFO (First In First Out) matching algorithm. The order that comes in first at a certain price level gets matched first, and the last one gets matched last.

There are many matching algorithms. These algorithms are commonly used in futures trading. For example, a FIFO with LMM (Lead Market Maker) algorithm allocates a certain quantity to the LMM based on a predefined ratio ahead of the FIFO queue, which the LMM firm negotiates with the exchange for the privilege. See more matching algorithms on the CME website [24]. The matching algorithms are used in many other scenarios. A typical one is a dark pool [25].

Determinism

There is both functional determinism and latency determinism. We have covered functional determinism in previous sections. The design choices we make, such as sequencer and event sourcing, guarantee that if the events are replayed in the same order, the results will be the same.

With functional determinism, the actual time when the event happens does not matter most of the time. What matters is the order of the events. In Figure 13.23, event timestamps from discrete uneven dots in the time dimension are converted to continuous dots,

and the time spent on replay/recovery can be greatly reduced.

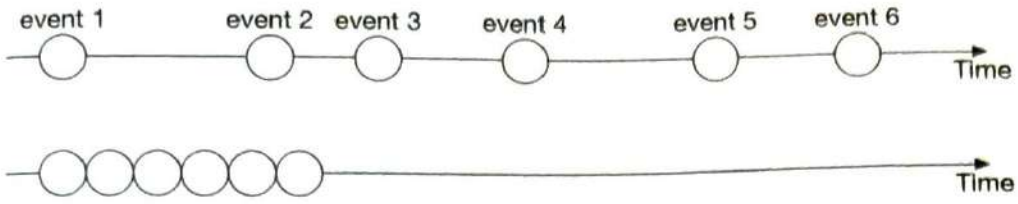


Figure 13.23: Time in event sourcing

Latency determinism means having almost the same latency through the system for each trade. This is key to the business. There is a mathematical way to measure this: the 99th percentile latency, or even more strictly, the 99.99th percentile latency. We can leverage HdrHistogram [26] to calculate latency. If the 99th percentile latency is low, the exchange offers stable performance across almost all the trades.

It is important to investigate large latency fluctuations. For example, in Java, safe points are often the cause. The HotSpot JVM [27] Stop-the-World garbage collection is a well-known example.

This concludes our deep dive on the critical trading path. In the remainder of this chapter, we take a closer look at some of the more interesting aspects of other parts of the exchange.

Market data publisher optimizations

As we can see from the matching algorithm, the L3 order book data gives us a better view of the market. We can get free one-day candlestick data from Google Finance, but it is expensive to get the more detailed L2/L3 order book data. Many hedge funds record the data themselves via the exchange real-time API to build their own candlestick charts and other charts for technical analysis.

The market data publisher (MDP) receives matched results from the matching engine and rebuilds the order book and candlestick charts based on that. It then publishes the data to the subscribers.

The order book rebuild is similar to the pseudocode mentioned in the matching algorithms section above. MDP is a service with many levels. For example, a retail client can only view 5 levels of L2 data by default and needs to pay extra to get 10 levels. MDP's memory cannot expand forever, so we need to have an upper limit on the candlesticks. Refer to the data models section for a review of the candlestick charts. The design of the MDP is in Figure 13.24.

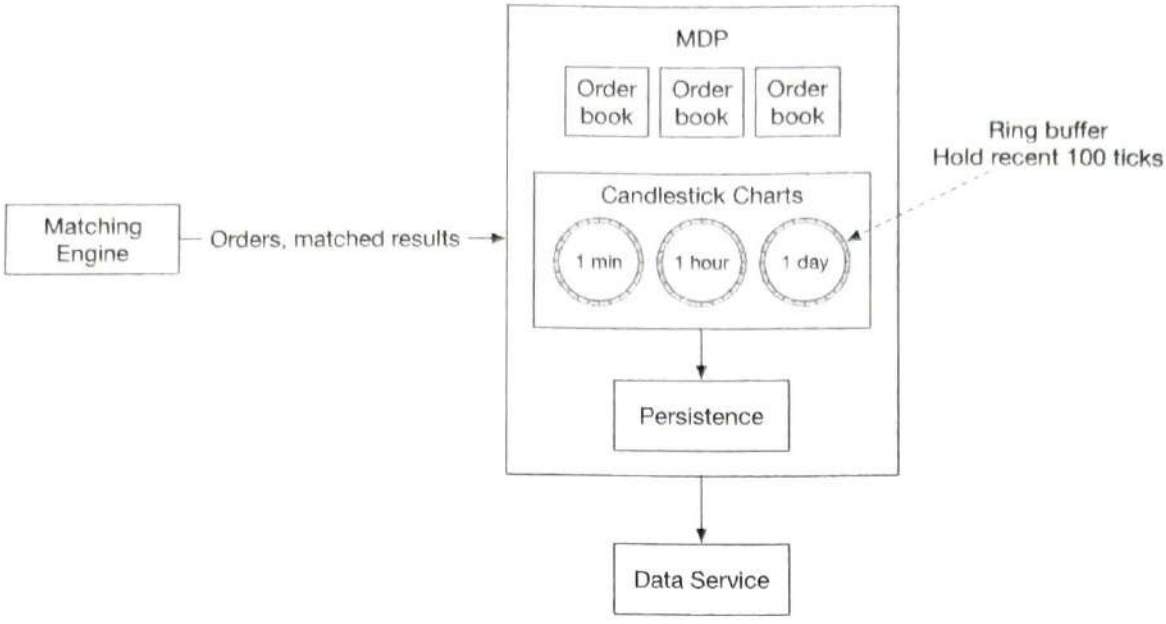


Figure 13.24: Market Data Publisher

This design utilizes ring buffers. A ring buffer, also called a circular buffer, is a fixed-size queue with the head connected to the tail. A producer continuously produces data and one or more consumers pull data off it. The space in a ring buffer is pre-allocated. There is no object creation or deallocation necessary. The data structure is also lock-free. There are other techniques to make the data structure even more efficient. For example, padding ensures that the ring buffer's sequence number is never in a cache line with anything else. Refer to [28] for more detail.

Distribution fairness of market data

In stock trading, having lower latency than others is like having an oracle that can see the future. For a regulated exchange, it is important to guarantee that all the receivers of market data get that data at the same time. Why is this important? For example, the MDP holds a list of data subscribers, and the order of the subscribers is decided by the order in which they connect to the publisher, with the first one always receiving data first. Guess what happens, then? Smart clients will fight to be the first on the list when the market opens.

There are some ways to mitigate this. Multicast using reliable UDP is a good solution to broadcast updates to many participants at once. The MDP could also assign a random order when the subscriber connects to it. We look at multicast in more detail.

Multicast

Data can be transported over the internet by three different types of protocols. Let's take a quick look.

1. Unicast: from one source to one destination.
2. Broadcast: from one source to an entire subnetwork.
3. Multicast: from one source to a set of hosts that can be on different subnetworks.

Multicast is a commonly-used protocol in exchange design. By configuring several receivers in the same multicast group, they will in theory receive data at the same time. However, UDP is an unreliable protocol and the datagram might not reach all the receivers. There are solutions to handle retransmission [29].

Colocation

While we are on the subject of fairness, it is a fact that a lot of exchanges offer colocation services, which put hedge funds or brokers' servers in the same data center as the exchange. The latency in placing an order to the matching engine is essentially proportional to the length of the cable. Colocation does not break the notion of fairness. It can be considered as a paid-for VIP service.

Network security

An exchange usually provides some public interfaces and a DDoS attack is a real challenge. Here are a few techniques to combat DDoS:

1. Isolate public services and data from private services, so DDoS attacks don't impact the most important clients. In case the same data is served, we can have multiple read-only copies to isolate problems.
2. Use a caching layer to store data that is infrequently updated. With good caching, most queries won't hit databases.
3. Harden URLs against DDoS attacks. For example, with an URL like `https://my.website.com/data?from=123&to=456`, an attacker can easily generate many different requests by changing the query string. Instead, URLs like this work better: `https://my.website.com/data/recent`. It can also be cached at the CDN layer.
4. An effective safelist/blocklist mechanism is needed. Many network gateway products provide this type of functionality.
5. Rate limiting is frequently used to defend against DDoS attacks.

Step 4 - Wrap Up

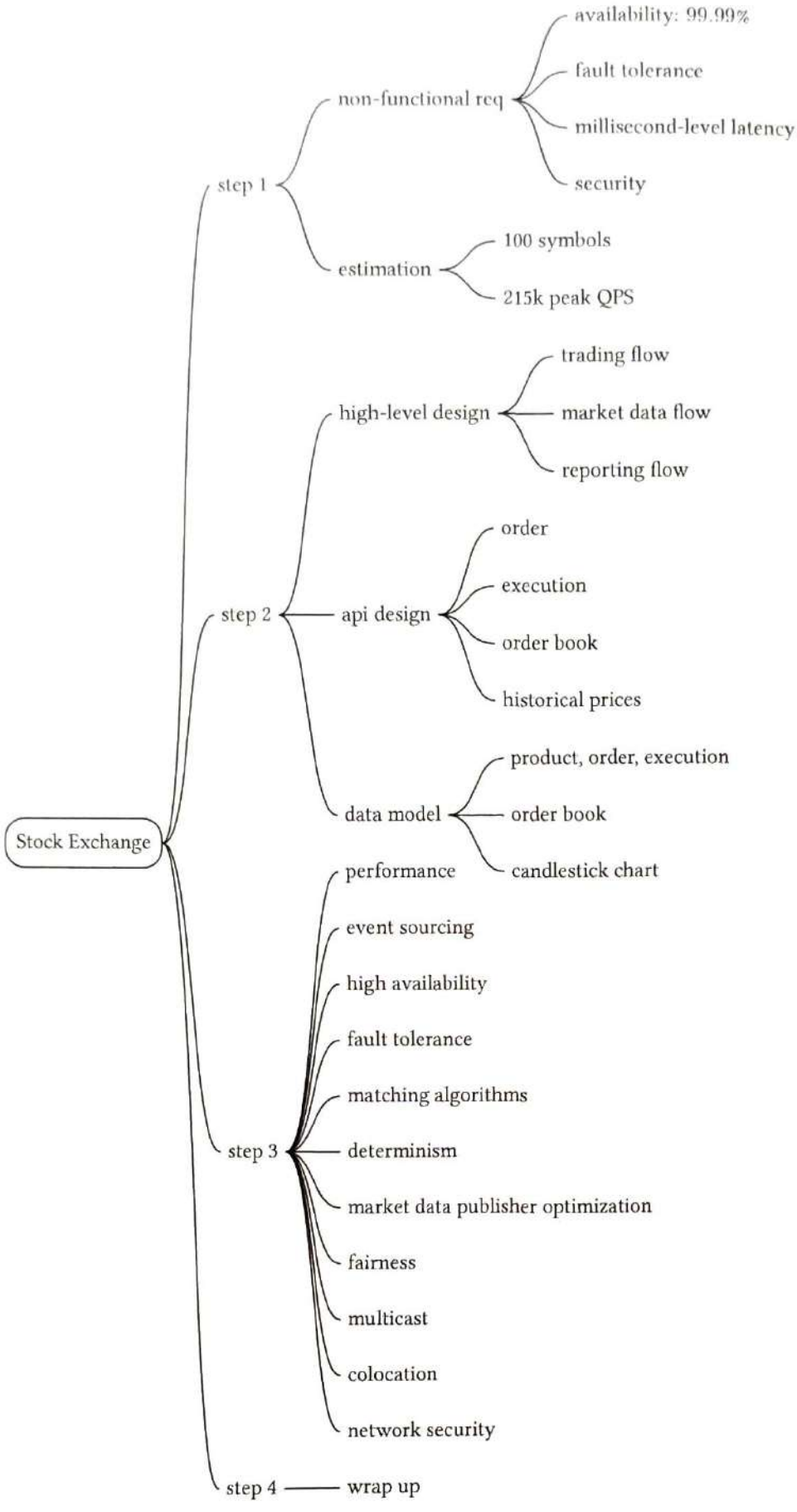
After reading this chapter, you may come to the conclusion that an ideal deployment model for a big exchange is to put everything on a single gigantic server or even one single process. Indeed, this is exactly how some exchanges are designed!

With the recent development of the cryptocurrency industry, many crypto exchanges use cloud infrastructure to deploy their services [30]. Some decentralized finance projects are based on the notion of AMM (Automatic Market Making) and don't even need an order book.

The convenience provided by the cloud ecosystem changes some of the designs and lowers the threshold for entering the industry. This will surely inject innovative energy into the financial world.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Chapter Summary



Reference Material

[1] LMAX exchange was famous for its open-source Disruptor. https://www.lmax.com/exchange.

[2] IEX attracts investors by "playing fair", also is the "Flash Boys Exchange". https://en.wikipedia.org/wiki/IEX.

[3] NYSE matched volume. https://www.nyse.com/markets/us-equity-volumes.

[4] HKEX daily trading volume. https://www.hkex.com.hk/Market-Data/Statistics/Consolidated-Reports/Securities-Statistics-Archive/Trading_Value_Volume_And_Number_Of_Deals?sc_lang=en#select1=0.

[5] All of the World's Stock Exchanges by Size. http://money.visualcapitalist.com/all-of-the-worlds-stock-exchanges-by-size/.

[6] Denial of service attack. https://en.wikipedia.org/wiki/Denial-of-service_attack.

[7] Market impact. https://en.wikipedia.org/wiki/Market_impact.

[8] Fix trading. https://www.fixtrading.org/.

[9] Event Sourcing. https://martinfowler.com/eaaDev/EventSourcing.html.

[10] CME Co-Location and Data Center Services. https://www.cmegroup.com/trading/colocation/co-location-services.html.

[11] Epoch. https://www.epoch101.com/.

[12] Order book. https://www.investopedia.com/terms/o/order-book.asp.

[13] Order book. https://en.wikipedia.org/wiki/Order_book.

[14] How to Build a Fast Limit Order Book. https://bit.ly/3ngMtEO.

[15] Developing with kdb+ and the q language. https://code.kx.com/q/.

[16] Latency Numbers Every Programmer Should Know. https://gist.github.com/jboner/2841832.

[17] mmap. https://en.wikipedia.org/wiki/Memory_map.

[18] Context switch. https://bit.ly/3pva7A6.

[19] Reliable User Datagram Protocol. https://en.wikipedia.org/wiki/Reliable_User_Datagram_Protocol.

[20] Aeron. https://github.com/real-logic/aeron/wiki/Design-Overview.

[21] Chaos engineering. https://en.wikipedia.org/wiki/Chaos_engineering.

[22] Raft. https://raft.github.io/.

[23] Designing for Understandability: the Raft Consensus Algorithm. https://raft.github.io/slides/uiuc2016.pdf.

[24] Supported Matching Algorithms. https://bit.ly/3aYoCEo.

[25] Dark pool. https://www.investopedia.com/terms/d/dark-pool.asp.

[26] HdrHistogram: A High Dynamic Range Histogram. http://hdrhistogram.org/.

[27] HotSpot (virtual machine). https://en.wikipedia.org/wiki/HotSpot_(virtual_machine).

[28] Cache line padding. https://bit.ly/3lZTFWz.

[29] NACK-Oriented Reliable Multicast. https://en.wikipedia.org/wiki/NACK-Oriented_Reliable_Multicast.

[30] AWS Coinbase Case Study. https://aws.amazon.com/solutions/case-studies/coinbase/.

Afterword

Congratulations! You have completed this interview guide. You have accumulated skills and knowledge with which to design complex systems. Not everyone has the discipline to do what you have done, to learn what you have learned. Take a moment to pat yourself on the back. Your hard work will pay off.

Landing your dream job is a long journey and requires lots of time and effort. Practice makes perfect. Best of luck!

Thank you for buying and reading this book. Without readers like you, our work would not exist. We hope you have enjoyed the book!

If you have comments or questions about this book, feel free to send us an email at hi@bytebytego.com. If you notice any errors, please let us know so we can make corrections for the next edition. Thank you!

Join the community

We created a members-only Discord group. It is designed for community discussions on the following topics:

- System design fundamentals.
- Showcasing design diagrams and getting feedback.
- Finding mock interview buddies.
- General chat with community members.

Come join us and introduce yourself to the community, today! Use the link below or scan the QR code.

http://bit.ly/systemdiscord

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Made in United States
North Haven, CT
20 September 2022

24375513R00239

This book is fantastic. It is a great continuation of the first book. I strongly recommend it to anyone who is studying for system design interviews.

– Sunny Patel, Software Engineering Manager at *Microsoft*

I was a Tech Lead at FAANG, but needed help in getting up to speed with unfamiliar domains. If you put in the work, this book will help you acquire the breadth and depth of knowledge you need for talking about bottlenecks and alternatives, as expected of a Tech Lead.

– Herbert Degano, Staff Software Engineer at *Coinbase*

What's inside?

- An insider's take on what interviewers really look for and why.
- A 4-step framework for solving any system design interview question.
- 13 real system design interview questions with detailed solutions.
- 303 diagrams to visually explain how different systems work.

About the Authors

Alex Xu is a software engineer and author. His book 'System Design Interview - An Insider's Guide (Volume 1)' is an Amazon bestseller, which has been translated into six languages. He has worked at Twitter, Apple, and Zynga.

Sahn Lam is a software engineer with decades of experience in building scalable systems at high-growth companies like Discord, Zynga, and NetApp. He has designed, built, operated, and optimized many interesting distributed systems used by millions of users.

You can connect with Alex on social media, where he shares system design interview tips every week.

twitter.com/alexxubyte

bit.ly/linkedinaxu

ISBN 9781736049112